*Information Systems* Vol. 16, No. 4, pp. 375–385, 1991
Printed in Great Britain. 

0306-4379/91 $3.00 + 0.00


# A FILE STRUCTURE FOR SEMANTIC DATABASES

NAPHTALI RISHE
School of Computer Science, Florida International University—The State University of Florida at Miami, University Park, Miami, FL 33199, U.S.A.

*(Received 25 November 1988; in final revised form 15 October 1990)*

**Abstract**—This paper presents a highly-efficient file structure for the storage of semantic databases. A low-level access language is presented, such that an arbitrary query can be performed as one or several elementary queries of the language. Most elementary queries, including such non-trivial queries as range queries and others, can be performed in just one single access to the disk.

*Key words:* Semantic databases, implementation, semantic binary model, file structure, query languages, transactions, indices, B-tree, efficiency, database access primitives, facts, inverted storage

## 1. INTRODUCTION

Since Abrial [1], many semantic data models have been studied in the computer science literature. Although somewhat different in their terminology and their selection of tools used to describe the semantics of the real world, they have several common principles:

- The entities of the real world are represented in the database in a manner invisible to the user. (Unlike that, in the relational model the entities are represented by the values of keys of some tables; in the network model the entities are represented by record occurrences.) Hereinafter, the user-invisible representations of real-world entities are referred to as "abstract objects". The "concrete objects", or "printable values", are numbers, character strings etc. The concrete objects have conventional representations on paper and in the computer.
- The entities are classified into types, or categories, which need not be disjoint. Meta-relations of inclusion are defined between the categories.
- Logically-explicit relationships are specified among abstract objects (e.g. "person $p_1$ is the mother of person $p_2$") and between abstract objects and concrete objects (e.g. "person $p_1$ has first name 'Jack"). There are no direct relationships among concrete objects. In most semantic models, only binary relations are allowed, since higher order relations do not add any power of semantic expressiveness [2–4], but do decrease the flexibility of the database and representablility of partially unknown information, and add complexity and potential for logical redundancy [4].

The advantages of the semantic models vs the relational and older models with respect to database design and maintenance, data integrity, conciseness of languages and ease of DML programming have been discussed in many works [e.g. 4]. This paper advocates the potential of an efficient implementation for the semantic models.

Several semantic data models have been implemented as interfaces to database management systems in other data models, e.g. the relational or the network model [5]. There are also less typical, direct implementations [e.g. 6–8]. The efficiency of an interface implementation is limited to that of the conventional DBMS, and is normally much worse due to the interface overhead. The direct implementations are commonly believed to have to be less time-efficient than the conventional systems, as a trade-off for the extra services that semantic databases provide. However, this author

contends that the semantic models have the potential for a much more efficient implementation than the conventional data models. This is due to two reasons:

- All the physical aspects of representation of information by data are invisible to the user in the semantic models. This creates a greater potential for optimization: more things may be changed for efficiency considerations, without affecting the user programs. The relational model has more data independence than the older models. For example, the order of rows in the tables (relations) is invisible to the user. The semantic models have even more data independence. For example, the representation of real-world entities by printable values is invisible to the user. One may recall that not long ago the relational model was criticized as less efficient than the network and hierarchical models. However, it is clear now that optimizing relational database systems have the potential to be much more efficient than the network and hierarchical system due to the data independence of the relational model.
- In the semantic models, the system knows more about the meaning of the user's data and about the meaningful connections between such data. This knowledge can be utilized to organize the data so that meaningful operations can be performed faster at the expense of less meaningful or meaningless operations.

In this paper, the author uses the semantic binary model (SBM) [3, 4, 9], a descendant of the model proposed in Ref. [1]. This model does not have as rich an arsenal of tools for semantic description as can be found in some other semantic models, e.g. the IFO model [10], SDM [11] (implementation [12]), the functional model [13] (implementation [7]), SEMBASE [14], NIAM [15, 16, 17], GEM [5], TAXIS [18] or the semi-semantic entity-relationship model [19]. Nevertheless, the SBM has a small set of sufficient simple tools by which all the semantic descriptors of the other models can be constructed. This makes SBM easier to use for the novice, easier to implement, and usable for delineation of the common properties of the semantic models. The results of this paper are practically independent of the choice of a particular semantic model, and therefore they apply to almost all of the other semantic models.

The semantic binary model represents the information of an application's world as a collection of elementary facts of two types: unary facts categorizing objects of the real world; and binary facts establishing relationships of various kinds between pairs of objects. The graphical database schema and the integrity constraints determine what sets of facts are meaningful, i.e. can comprise an instantaneous database (the database as may be seen at some instance of time.)

*Example 1*

Consider a database of which the following is a sub-schema:

- Category *COMPANY*
- Category *PRODUCT*
- Relation *company-name* from *COMPANY* to the category of values *String* (1 : 1)
- Relation *description* from *PRODUCT* to the category of values *String* (1 : 1)
- Relation **manufactures** from *COMPANY* to *PRODUCT* (*m* : *m*)

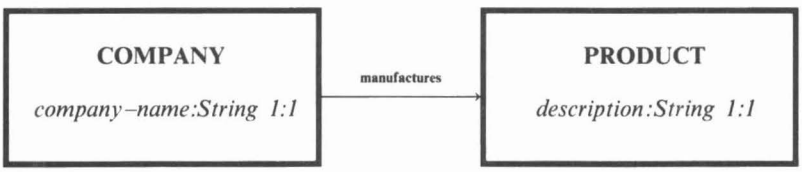


Fig. 1. A sub-schema of a database.

The following set of facts can be a part of a logical instantaneous database:

1. object1 **COMPANY**
2. object1 ***COMPANY-NAME*** 'IBM'

3. object1 ***MANUFACTURED*** object2
4. object1 ***MANUFACTURED*** object3
5. object2 ***PRODUCT***
6. object2 ***DESCRIPTION*** 'IBM/SYSTEM-2'
7. object3 ***PRODUCT***
8. object3 ***DESCRIPTION*** 'MONOCHROMATIC-MONITOR'

The formal semantics of the semantic binary model is defined in Ref. [20] using the methodology proposed in Ref. [21]. The syntax and informal semantics of the model and its languages (data definition languages, 4th generation data manipulation languages, non-procedural languages for queries, updates, specification of constraints, userviews etc.) are given in Ref. [4]. A non-procedural semantic database language of maximal theoretically-possible expressive power is given in Ref. [22] (in this language, one can specify every computable query, transaction, constraint etc.)

The following section proposes an efficient storage structure for the SBM.

## 2. STORAGE STRUCTURE

### *2.1. Abstracted Level*

Every abstract object in the database is represented by a unique integer identifier. The categories and relations of the schema are also treated as abstract objects and hence have unique identifiers associated with them. Information in the database can then be represented using two kinds of facts, denoted $xC$ and $xRy$, where $x$ is the identifier associated with an abstract object, $C$ and $R$ are the identifiers associated with a category or a relation, respectively, and $y$ is either an identifier corresponding to an abstract object or a concrete object (a number or a text string). $xC$ indicates that the object $x$ belongs to the category $C$. $xRy$ indicates that the object $x$ is associated with the object $y$ by the relation $R$. Logically, the instantaneous database is a set of such facts.

### *2.2. Goals*

#### *2.2.1. Efficiency of retrieval requests*

AT the intermediate level of processing queries and program retrieval requests, the queries are decomposed into *atomic retrieval operations* of the types listed below. The primary goal of the physical file structure is to allow a very efficient performance for each of the atomic requests. Namely, *each atomic retrieval request normally requires only one disk access*, provided the output information is small enough to fit into one block. When the output is large, the number of blocks retrieved is close to the minimal number of blocks needed to store the output information.

| | |
|---|---|
| (1) **aC** | Verify the fact $aC$. (For a given abstract object $a$ and category $C$, verify whether the object $a$ is in the category $C$.) |
| (2) **aRy** | Verify the fact $aRy$. |
| (3) **a?** | For a given abstract object $a$, find all the categories to which $a$ belongs. |
| (4) **?C** | For a given category, find its objects. |
| (5) **aR?** | For a given abstract object $a$ and relation $R$, retrieve all $y$ such that $aRy$. (The objects $y$ may be abstract or concrete.) |
| (6) **?Ra** | For a given abstract object $a$ and relation $R$, retrieve all abstract objects $x$ such that $xRa$. |
| (7) **a? + a?? + ??a** | Retrieve all the immediate information about an abstract object. (That is, for a given abstract object $a$, retrieve all of its direct and inverse relationships, i.e. the relations $R$ and objects $y$ such that $aRy$ or $yRa$; and the categories to which $a$ belongs.) (Although this request can be decomposed into a series of requests of the previous types, we wish to be able to treat it separately in order to ensure that the whole request will normally be performed in a single disk access. This will also allow a single-access performance of requests which require |

several, but not all, of the facts about an object, e.g. a query to find the first name, the last name, and the age of a given person.)

(8) **?Rv** For a given relation (attribute) $R$ and a given concrete object (value) $v$, find all abstract objects $x$ such that $xRv$.

(9) **?R[v1, v2]** For a given relation (attribute) $R$ and a given range of concrete objects $[v_1, v_2]$, find all objects $x$ and $v$ such that $xRv$ and $v_1 \leqslant v \leqslant v_2$. (The comparison "$\leqslant$" is appropriate to the type of $v$).

The elementary queries defined above form a lower-level language of retrieval from semantic databases. Any query in any language can be solved by performing several elementary queries and processing their results in the memory.

*Example 2.1*

Consider the following query in the semantic predicate calculus:

**get** c.NAME, c.ADDRESS
**where** c **is an** *LTD-COMPANY* **and** c.YEAR-FOUNDED < 1989 **and**
**exists** p in *PRODUCT*: c *MANUFACTURES* p **and** p.COST >= 670 **and**
p.COST <= 680

(This query prints the names and addresses of the limited companies founded before 1989 that manufacture products costing between \$670 and \$680. It is assumed that *LTD-COMPANY* is a subcategory of *COMPANY*.)

A query processor/optimizer can perform this as follows:

(1) Perform (? COSTS [670, 680]) (resulting, say, in objects $p_1$, $p_2$, and $p_3$.)
(2) For each of $i$ in 1 . . . 3 perform (? MANUFACTURES $p_i$) (let us assume that the union of the results of the three queries is $c_1$, $c_2$, $c_3$ and $c_4$).
(3) For each $j$ in 1 . . . 4 perform the elementary ($c_j? + c_j?? + ??c_j$), obtaining the immediate information about the company $c_j$. This includes the information necessary to check (c.YEAR-FOUNDED < 1989 **and** c **is an** *LTD-COMPANY*) as well as the values of *NAME* and *ADDRESS* to be printed if the result of the latter is positive.

The total number of elementary queries here was 8.

### 2.2.2. *Efficiency of update transactions*

Efficient performance of update transactions is required, although more than one disk access per transaction is allowed.

A transaction is a set of interrelated update requests to be performed as one unit. Transactions are generated by programs and by interactive users. A transaction can be generated by a program fragment containing numerous update commands, interleaved with other computations. However, until the last command within a transaction is completed, the updates are not physically performed, but rather accumulated by the DBMS. Upon completion of the transaction, the DBMS checks its integrity and then physically performs the update. The partial effects of the transaction may be inconsistent. Every program and user sees the database in a consistent state: until the transaction is committed, its effects are invisible.

A completed transaction is composed of a set of facts to be deleted from the database, a set of facts to be inserted into the database, and additional information needed to verify that there is no interference between transactions of concurrent programs. If the verification produces a positive result, then the new instantaneous database is: [(the-old-instantaneous-database) − (the-set-of-facts-to-be-deleted)]∪(the-set-of-facts-to-be-inserted).

*Example 2.2*

Consider the database of *Example 1.*

The following is a transaction to rename *Burroughs* into *Unisys*, transfer all business from *Sperry* to *Unisys*, and delete *Sperry*.

```
transaction
 for b where s COMPANY-NAME 'Burroughs' do
  for s where s COMPANY-NAME 'Sperry' do
   begin
    b.COMPANY-NAME:='Unisys';
    for p where s MANUFACTURES p do
          relate b MANUFACTURES p;
    decategorize s from COMPANY
    end
```

Let us assume that before the transaction the two companies are objects $b_0$ and $s_0$, respectively and Sperry manufactures products $p_1$ and $p_2$. The following queries were performed from within this transaction:

(a) ? *COMPANY-NAME* 'Burroughs' (results in $b_0$)
(b) ? *COMPANY-NAME* 'Sperry' (results in $s_0$)
(c) $s_0$ *MANUFACTURES*? (results in $\{p_1, p_2\}$)

At the end of the programmatic transaction, the accumulated transaction will be $(V, D, I)$, where $V$, the verification specification, is the above three queries with their time-stamps; $D$ is the following specification of the facts to be deleted:

$s_0$ *COMPANY*
$s_0$ *COMPANY-NAME*$\star$
$s_0$ *MANUFACTURES*$\star$
$b_0$ *NAME*$\star$

and $I$ is the following set of facts to be inserted:

$b_0$ *NAME* 'Unisys'
$b_0$ *MANUFACTURES* $p_1$
$b_0$ *MANUFACTURES* $p_2$

### *2.3. Solution: a File Structure Achieving the Goals*

The following file structure supports the above requirements. The entire database is stored in a single file. This file contains all the facts of the database ($xC$ and $xRy$) and additional information, called *inverted facts*, which are described below. The file is maintained as a B-tree. The variation of the B-tree used here 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 the facts and inverted facts.

The inverted facts do introduce some physical redundancy (no logical redundancy since they are invisible to the user), which results in a storage overhead and update-time overhead. However, as it is shown below, this overhead is not greater than if index structures were used. Of course, it is impossible to achieve any reasonable retrieval efficiency without physical redundancy, such as the indices in conventional implementations or the inverted facts proposed in this paper.

The facts which are close to each other in the lexicographic order reside close together in the file. (Notice, that although technically the B-tree-key is the entire fact, it is of varying length and on the average is only several bytes long, which is the average size of the encoded fact $xRy$). The total size of the data stored in the index-level blocks of the B-tree is $<1\%$ of the size of the database: e.g. each 10,000-byte data block may be represented in the index level by its first fact—5 bytes—and block address—3 bytes—which would amount to 0.08% of the data block. Thus, all the index blocks will fit into even a relatively small main memory.

The file contains the original facts and additionally the following "inverted facts":

1. In addition to $\boldsymbol{xC}$, we store its inverse $\bar{\boldsymbol{C}}\boldsymbol{x}$. ($\bar{C}$ is the system-chosen identifier to represent the inverse information about the category $C$. For example, it can be defined as $\bar{C} = 0\text{–}C$.) (If a category $C_1$ is a subcategory of category $C_2$, an object $a$ belongs to $C_1$ and, thus, also to $C_2$,

then we choose to store both inverted facts $\overline{C_1}a$ and $\overline{C_2}a$. When the user requests the deletion of the fact $aC_2$, it triggers automatic deletion of the facts $aC_1$, $\overline{C_1}a$, and $\overline{C_2}a$ in order to guarantee consistency.) Thus, the elementary query ?C to find all the objects of the category $C$, can be answered by examining the (inverted) facts whose prefix is $\bar{C}$. The latter inverted facts are clustered together in the lexicographic order of the physical database.

2. In addition to $\boldsymbol{xRv}$, where $v$ is a concrete object (a number, a string, or a value of other type), we store $\boldsymbol{\bar{R}vx}$. Thus, the range query "?R[$v_1$, $v_2$]" is satisfied by all and only the inverted facts which are positioned in the file between $\bar{R}v_1$ and $\bar{R}v_2$HighSuffix. (HighSuffix is a suffix which is lexicographically greater than any other possible suffix.) Thus, the result will most probably appear in one physical block, if it can fit into one block.
3. In addition to $\boldsymbol{xRy}$, where both $x$ and $y$ are abstract objects, we store $\boldsymbol{y\bar{R}x}$. Thus, for any abstract object $x$, all its relationships $xRy$, $xRv$, $zRx$, and $xC$ can be found in one place in the file: the regular and inverted facts which begin with the prefix $x$. (The infixes are: categories for $xC$, relations for $xRy$ and $xRv$, and inverse relations $x\bar{R}z$ from which we find $z$ such that $zRx$.)

*Example 2.3*

Consider the instantaneous database of *Example 1*. The additional inverted facts stored in the database are:

1. ***COMPANY***$^{inv}$ object1
2. ***COMPANY-NAME***$^{inv}$ 'IBM' object1
3. object2 ***MANUFACTURED***$^{inv}$ object1
4. object3 ***MANUFACTURED***$^{inv}$ object1
5. ***PRODUCT***$^{inv}$ object2
6. ***DESCRIPTION***$^{inv}$ 'IBM/SYSTEM-2' object2
7. ***PRODUCT***$^{inv}$ object3
8. ***DESCRIPTION***$^{inv}$ 'MONOCHROMATIC-MONITOR' object3

Notice that facts $xRa$ and $xRv$ ($x$ and $a$ are abstract objects, $v$ is a value) are inverted dissimilarly. This is because we have different types of atomic retrieval requests concerning abstract and concrete objects:

- Concrete objects can be used to form range queries, e.g. "Find all persons with salaries between \$40,000 and \$50,000". In such queries we know the identifier of the relation and partial information about the value. Therefore we need to use the inverted facts with $\bar{R}$ as the prefix. Unlike concrete objects, ranges of abstract objects cannot form a meaningful range query.
- On the other hand, we have multiple-fact retrievals about an abstract object, e.g. "Find all the immediate information about a given person $p$" (while such a request about a concrete object would be meaningless: "Find all the information about the number 5" makes no sense, as opposed to a meaningful query "Find information about item(s) whose price is \$5".) Here we know the object, but do not know the identifiers of the inverted relations. We need to cluster together all the inverted relations of one object. Therefore, the inverted relation should appear in the infix.

*Example 2.4*

When the set of original facts is interleaved and lexicographically sorted with the inverted facts of the previous example, we obtain:

1. object1 ***COMPANY***
2. object1 ***COMPANY-NAME*** 'IBM'
3. object1 ***MANUFACTURED*** object2
4. object1 ***MANUFACTURED*** object3
5. object2 ***DESCRIPTION*** 'IBM/SYSTEM-2'
6. object2 ***PRODUCT***
7. object2 ***MANUFACTURED***$^{inv}$ object1
8. object3 ***DESCRIPTION*** 'MONOCHROMATIC-MONITOR'
9. object3 ***PRODUCT***

10. object3 ***MANUFACTURED***$^{inv}$ object1
11. ***COMPANY***$^{inv}$ object1
12. ***DESCRIPTION***$^{inv}$ 'IBM/SYSTEM-2' object2
13. ***DESCRIPTION***$^{inv}$ 'MONOCHROMATIC-MONITOR' object3
14. ***COMPANY-NAME***$^{inv}$ 'IBM' object1
15. ***PRODUCT***$^{inv}$ object2
16. ***PRODUCT***$^{inv}$ object3

*Example 2.5*

To answer the elementary query to find all the information about object3, including its direct and inverse relationships, we find all the entries whose prefix is object3. These entries are clustered together in the sorted order.

To answer the elementary query "Find all objects manufactured by object1" we find all the facts whose prefix is object 1_***MANUFACTURED***. ('_' denotes concatenation.) These entries are clustered together in the sorted order.

The query to print the descriptions of the objects manufactured by the companies whose names are between 'IATA' and 'K-mart', we solve several elementary subqueries:

1. Find the companies whose names are in the range between the above strings. (We search for inverted facts which are lexicographically between ***COMPANY-NAME***$^{inv}$_'IATA' and ***COMPANY-NAME***$^{inv}$_'K-mart'_HighSuffix. For the instantaneous database given in the previous example we find only one inverted fact ***COMPANY-NAME***$^{inv}$_'IBM'_object1. The suffix object1 is the company we are looking for.)
2. Find the products manufactured by object1. (From facts with prefix object1_***MANUFACTURED*** we find suffixes {object2, object3}.)
3. Find the description of object2. (Prefix object2_***DESCRIPTION***);
4. Find the description of object3. (Prefix object3_***DESCRIPTION***);

The sorted file is maintained in a structure similar to a B-tree. The "records" of the B-tree are the regular and inverted facts. The records are of varying length. The B-tree-keys of the "records" are normally the entire B-tree-records, i.e. facts, regular and inverted. (An exception to this is when the record happens to be very long. The only potentially long records represent facts $xRv$ where $v$ is a very long character string. We employ a special handling algorithm for very long character strings.) Access to this B-tree does not require knowledge of the entire key: any prefix will do. All the index blocks of the B-tree can normally be held in cache.

At the most physical level, the data in the facts is compressed to minimal space. Also, since many consecutive facts share a prefix (e.g. an abstract object identifier) the prefix need not be repeated for each fact. In this way the facts are compressed further. The duplication in the number of facts due to the inverses is 100%, since there is only one inverse per each original fact [with a rare exception of the storage of redundant inverses of supercategories as described in (1)]. The B-tree causes additional 30% overhead. (This overhead occurs because in a B-tree the data blocks are only 75% full on the average, though this can be improved by periodic reorganization. The overhead for the index blocks of the B-tree is no more than 1–2% since they contain only one short fact per every data block.) The total space used for the database is therefore only about 160% more than the amount of information in the database, i.e. the space minimally required to store the database in the most compressed form with no regard to the efficiency of data retrieval or update. Thus, the date structure described herein is more efficient in space and time than the conventional approach with separate secondary index files for numerous fields.

No separate index files are needed for the file structure proposed in this paper. The duplication of data (i.e. inverted relations) together with the primary sparse index which is a part of the B-tree effectively eliminate the need for secondary (dense) indices. Furthermore, it eliminates the horrendous I/O operations caused by sequentially retrieving along a secondary index, since the sequence of information represented by our primary sparse index is also stored in consecutive physical locations. These claims are proven in the following section.

### *2.4. Proof of Time-efficiency of the File Structure*

**Lemma 1.** Let the file be logically perceived as a lexicographically ordered sequence of facts. Let *req* be an atomic retrieval request. Then there is a contiguous segment in the sequence, so that:

(i) all the facts in the segment satisfy the request *req*;
(ii) no fact outside the segment satisfies the *req*; and

(iii) the boundaries $fact_{start}$ and $fact_{end}$ of the segment can be derived from the syntax of the request *req*. (Thus, all the output facts are lexicographically between $fact_{start}$ and $fact_{end}$. The boundaries may be inclusive or exclusive.)

*Proof of Lemma 1.* The following are the ranges for the segments for each of the atomic requests (the symbol $\underline{HIGH}$ denotes the bit string "1111111111111....." which is lexicographically greater than any possible suffix in a fact):

| Request | Segment |
|---|---|
| 1. **aC** | $aC \leqslant fact \leqslant aC$<br>(Verify the fact $aC$.) |
| 2. **aRy** | $aRy \leqslant fact \leqslant aRy$<br>(Verify the fact $aRy$.) |
| 3. **a?** | $aC_{\min} \leqslant fact \leqslant aC_{\max}$ (Here, it is assumed that all the categories of the schema are enumerated by identifiers between $C_{\min}$ and $C_{\max}$.)<br>(For a given abstract object $a$, find what categories the object belongs to). |
| 4. **?C** | $\bar{C} < fact \leqslant \bar{C}\underline{HIGH}$<br>(For a given category, find its objects.) |
| 5. **aR?** | $aR < fact \leqslant aR\underline{HIGH}$<br>(For a given abstract object $a$ and relation $R$, retrieve all $y$ such that $aRy$.) |
| 6. **?Ra** | $a\bar{R} < fact \leqslant a\bar{R}\underline{HIGH}$<br>(For a given abstract object $a$ and relation $R$, retrieve all abstract objects $x$ such that $xRa$.) |
| 7. **a? + a?? + ??a** | $a < tuple \leqslant a\underline{HIGH}$<br>(Retrieve all the immediate information about an abstract object.) |
| 8. **?Rv** | $\bar{R}v < fact \leqslant \bar{R}v\underline{HIGH}$<br>(For a given relation (attribute)$R$ and a given concrete object $v$, find all abstract objects $x$ such that $xRy$.) |
| 9. **?R[v1, v2]** | $\bar{R}v_1 < fact \leqslant \bar{R}v_2\underline{HIGH}$<br>(For a given relation $R$ and a given range of concrete objects $[v_1, v_2]$, find all objects $x$ and $v$ such that: $xRv$ and $v_1 \leqslant v \leqslant v_2$.) |

• End of Lemma 1 •

In the following, an estimate is given for the number of disk accesses per atomic retrieval request. Two cases are considered:

*(A) One disk access per request of small output.* In the predominant case, the amount of information to be output for a given atomic request is much less than one block. According to Lemma 1, all the information to be output comprises one contiguous segment. The segment has as many facts as there are items to be output. Therefore, the segment is much less than one block. (In the physical storage the facts are prefix-compressed, so that the physical space for each fact is normally just a few bytes.) Hence, normally the segment fits into one block. We can find the address of this block in the cache-resident B-tree index with the key $fact_{start}$. Then, in a single access the block is brought to the memory. There is a small probability that the segment appears on the boundary of two blocks. In the latter case we may have to bring two blocks into the memory. On the other hand, the desired block[s] may have already been in cache and, thus, sometimes zero accesses are sufficient.

Thus, the retrieval efficiency for the atomic requests is the optimum, or very close to the optimum. (One cannot retrieve a memory-unavailable datum in less than one disk access.)

*(B) For large output, the efficiency is also close to the optimum.* When the output is larger than a block, so is the segment. If the output can be squeezed into $n$ blocks, then $n$ would be the theoretical optimum (not obtainable in any practical system) for the number of disk accesses per

request. All the facts of the segment can be squeezed into slightly more than $n$ blocks (depending on how much of the prefix can be compressed), say $1.1n$ blocks. Due to the space maintenance policy of the B-tree, each physical data block is 75% full on the average. The segment may begin in the middle of one block and end in another, thus, on the average, one additional block has to be fetched (the actual overhead of this type ranges between 0 and 2 block fetches). Thus the total expected number of blocks to be fetched is $(1.1n/0.75) + 1 = 1.47n + 1$. The number of disk accesses may be even less than that if some of the blocks are in cache.

## 3. COMPARISON TO PERFORMANCE OF IMPLEMENTATIONS OF THE RELATIONAL MODEL

The system proposed herein is not less efficient, and is normally more efficient, in both time and storage space than the relational model's implementations with multiple dense indices.

Let us consider a simple relational database composed of one relation T with attributes $A_1, A_2, \ldots, A_n$. Let us assume that for each $j$ there are queries of the type

$$\textbf{get } A_i \textbf{ where } A_j = c \qquad \text{(Q1)}$$

and that each of those queries is required to be performed in a reasonable time.

For the purpose of physical implementation, the relational model's databases can be technically represented (without affecting the user) as certain semantic binary databases. Specifically, the above relational schema can be regarded in the SBM as a category T and relations $A_i$ between the objects of T and values.

To assure reasonable time performance in the relational model for each of the above queries, we need a dense index on each of the attributes $A_i$. There are $n$ index files (or $n$ indices combined in one file in some implementations). The total size of the indices thus exceeds the size of the table T itself. Therefore the space overhead in the relational model is $>100\%$ and, thus, is greater than the space overhead in the proposed semantic implementation. Also, in the semantic implementation there is only one physical file, while there are many physical files in the relational implementations (and in some implementations there are as many files as *number_of_tables* $\times$ (1 + *number_of_attributes_per_table*). The management of multiple files is not only a hassle but also contributes to additional space overhead due to allocation of growth areas for each file.

With respect to the time required to solve the simple queries of type Q1 it is the same in the best relational implementations and in the proposed semantic implementation. Namely, the time is

$$(1 + \textit{number\_of\_values\_in\_the\_output}) \times \textit{time\_to\_retrieve\_one\_block}.$$

(In the relational implementation, there will be one visit to the dense index on $A_j$, and for every $A_j = c$ found there, there will be one random access to the main table. In the semantic implementation, first the sub-query $?A_jc$ will be solved, and then for every match $x$ found the sub-query $xA_i?$ will be evaluated.)

If in Q1 we desired to print many attributes $A_i$ instead of just one, the same time results would be obtained in both implementations. Notice that in the semantic implementation proposed herein all the immediate information of an object, including all its attributes, is clustered together.

Now let us consider updates. Insertion of a row into the relational table takes replacement of one block in the main table and $n$ blocks in the dense indices. In the semantic implementation there is insertion of the primary facts about the new object $ob$: $obA_1c_1, \ldots, obA_nc_n$ (all the primary facts will appear in contiguous storage in one block) and $n$ inverse facts in possibly $n$ different blocks. Thus, here, as well as in the other types of sample updates, the performance of the semantic implementation is not worse than that of the relational implementations supporting efficiency of queries.

The advantages in the schematic implementation's performance become even more significant for more complex queries and updates. Though the detailed analysis of these is beyond the space-limit of this paper, I would like to mention that, for example, queries requiring natural join in the relational implementations would be more efficient in the semantic implementation because there are direct explicit relationships between the categories instead of relationships represented implicitly by foreign keys in the relational model. The gap in performance between the faster

semantic implementation and the relational implementations is even greater when the relational keys are composed of more than one attribute and when the relationships between the tables are many-to-many, which requires an extra table to represent the many-to-many relationship in the relational implementations. The gap increases with the number of joins in the query. In general, the advantage in the efficiency of the proposed semantic implementation versus the relational implementations normally becomes greater as the complexity of the queries increases.

Of course, there are also major efficiency advantages in the semantic implementation in support of semantic complexities of the real world, which are very awkwardly and inefficiently implemented in the relational implementations. These complexities include intersecting categories, sub-categories, categories with no keys, varying-length attributes, missing ("null") values, multiple values, etc.

## 4. CONCLUSION

We have implemented this data structure in a prototype DBMS at the University of California, Santa Barbara [23, 24]. Further improved software is under development. Our implementation allows single-processor multi-user parallel access to the database. Optimistic concurrency control is used.

Although the best results are obtained from our DBMS for the SBM it can also be used efficiently with all other major semantic and conventional database models. This is due to the fact that the relational, network and hierarchical data models can be implemented via the SBM (as shown in Ref. [4]).

Currently, at Florida International University, we are working on a project, financed by the state government, to extend our semantic DBMS implementation into a massively-parallel very-high-throughput database machine [25], to be composed of many [thousand(s)] processors, each equipped with a permanent storage device and a large cache memory. Our analysis has shown that the proposed file structure greatly increases the parallelism in the operations of the DBMS, which can be utilized by large-scale parallel machines.

*Acknowledgements*—The author gratefully acknowledges the advice of Narayanan Vijaykumar, Li Qiang, Nagarajan Prabhakaran, Doron Tal, David Barton and Scott Graham. This research has been supported in part by a grant from the Florida High Technology and Industry Council.

## REFERENCES

[1] J. R. Abrial. Data semantics. In *Data Base Management* (J. W. Klimbie and K. L. Koffeman, Eds). North-Holland, Amsterdam (1974).

[2] G. Bracchi, P. Paolini and G. Pelagatti. Binary logical associations in data modelings. In *Modeling in Data Base Management Systems* (G. M. Nijssen, Ed.) *IFIP Work. Conf. on Modeling in DBMS's* (1976).

[3] N. Rishe. On representation of medical knowledge by a binary data model. *J. Math. Comput. Modelling* **8**, 623–626 (1987).

[4] N. Rishe. *Database Design Fundamentals: a Structured Introduction to Databases and a Structured Database Design Methodology*. Prentice-Hall, Englewood Cliffs, NJ (1988).

[5] S. Tsur and C. Zaniolo. An implementation of GEM—supporting a semantic data model on a relational backend. In *Proc. ACM SIGMOD Int. Conf. on Management of Data* (1984).

[6] Y. E. Lien, J. E. Shopiro and S. Tsur. DSIS—A database system with interrelational semantics. In *Proc. 7th Int. Conf. on Very Large Data Bases* (C. Zaniolo and C. Delobel, Eds), pp. 465–477. IEEE Computer Society Press (1981).

[7] A. Chan, Sy. Danberg, S. Fox, W-T. K. Lin, A. Nori and D. R. Ries. Storage and access structures to support a semantic data model. In *Proc 8th Int. Conf. on Very Large Data Bases*. IEEE Computer Society Press (1982).

[8] R. L. Benneworth, C. D. Bishop, C. J. M. Turnbull, W. D. Holman and F. M. Monette. The implementation of GERM, an entity-relationship data base management system. In *Proc. 7th Int. Conf. on Very Large Data Bases* (C. Zaniolo and C. Delobel, Eds), pp. 465–477. IEEE Computer Society Press (1981).

[9] N. Rishe. Semantic database management: from microcomputers to massively parallel database machines. Keynote paper. In *Proc. 6th Symp. Microcomput. Microprocess. Applications*, Budapest, pp. 1–12 (1989).

[10] S. Abiteboul and R. Hull. IFO: a formal semantic database model. In *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems* (1984).

[11] M. Hammer and D. McLeod. Database description with SDM: a semantic database model. *ACM Trans. Database Systems* **6**(3), 351–386 (1981).

[12] D. Jagannathan, R. L. Guck, B. L. Fritchman, J. P. Thompson and D. M. Tolbert. SIM: a database system based on semantic model. In *Proc. SIGMOD Int. Conf. on Management Data*. ACM-Press, Chicago, IL (1988).

[13] D. W. Shipman. The functional data model and the data language DAPLEX. *ACM Trans. Database Systems* **6**(1), 140–173 (1981).

[14] R. King. SEMBASE: a semantic DBMS. In *Proc. Ist Workshop on Expert Database Systems*, Univ. South Carolina, Columbia, pp. 151–171 (1984).
[15] G. M. Nijssen. An architecture for knowledge base systems. In *Proc. SPOT-2 Conf.*, Stockholm (1981).
[16] G. M. A. Verheijen and J. Van Bekkum. NIAM—an information analysis method. In *Information Systems Design Methodologies: a Comparative Review* (T. W. Olle *et al.*, Eds.). IFIP, North-Holland, Amsterdam (1982).
[17] C. M. R. Leung and G. M. Nijssen. From a NIAM conceptual schema into the optimal SQL relational database schema. *Aust. Comput. J.* **19**(2),
[18] B. Nixon, L. Chung, I. Lauzen, A. Borgida and M. Stanley. Implementation of a compiler for a semantic data model: experience with taxis. In *Proc. ACM SIGMOD Conf.*, San Francisco, CA (1987).
[19] P. P. Chen. The entity-relationship model: toward a unified view of data. *ACM Trans. Database Systems 1* **1**, 9–36 (1976).
[20] N. Rishe. Database semantics. Tech. Rep. TRCS87-2, Univ. of California, Santa Barbara, CA (1987).
[21] N. Rishe. On denotational semantics of data bases. In *Lecture Notes in Computer Science*, Vol. 239. *Mathematical Foundations of Programming Semantics* (A. Melton, Ed.), pp. 249–274. Springer, New York (1986).
[22] N. Rishe. Postconditional semantics of data base queries. In *Lecture Notes in Computer Science*, Vol. 239 *Mathematical Foundations of Programming Semantics* (A. Melton, Ed.), pp. 275–295. Springer, New York (1986).
[23] N. Vijaykumar. Toward the Implementation of a DBMS based on the semantic binary model. M.S. Thesis, Univ. California, Santa Barbara, CA (1987).
[24] A. Jain. Design of a binary model based DBMS and conversion of binary model based schema to an equivalent schema in other major database models. M.S. Thesis, Univ. California, Santa Barbara, CA (1987).
[25] N. Rishe, D. Tal and Q. Li. Architecture for a massively parallel database machine. *Microprocess. Microprog. (Euromicro J.)* ***25****, 33–38 (1989).*