LBL-21269
Preprint c.2

Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA, BERKELEY

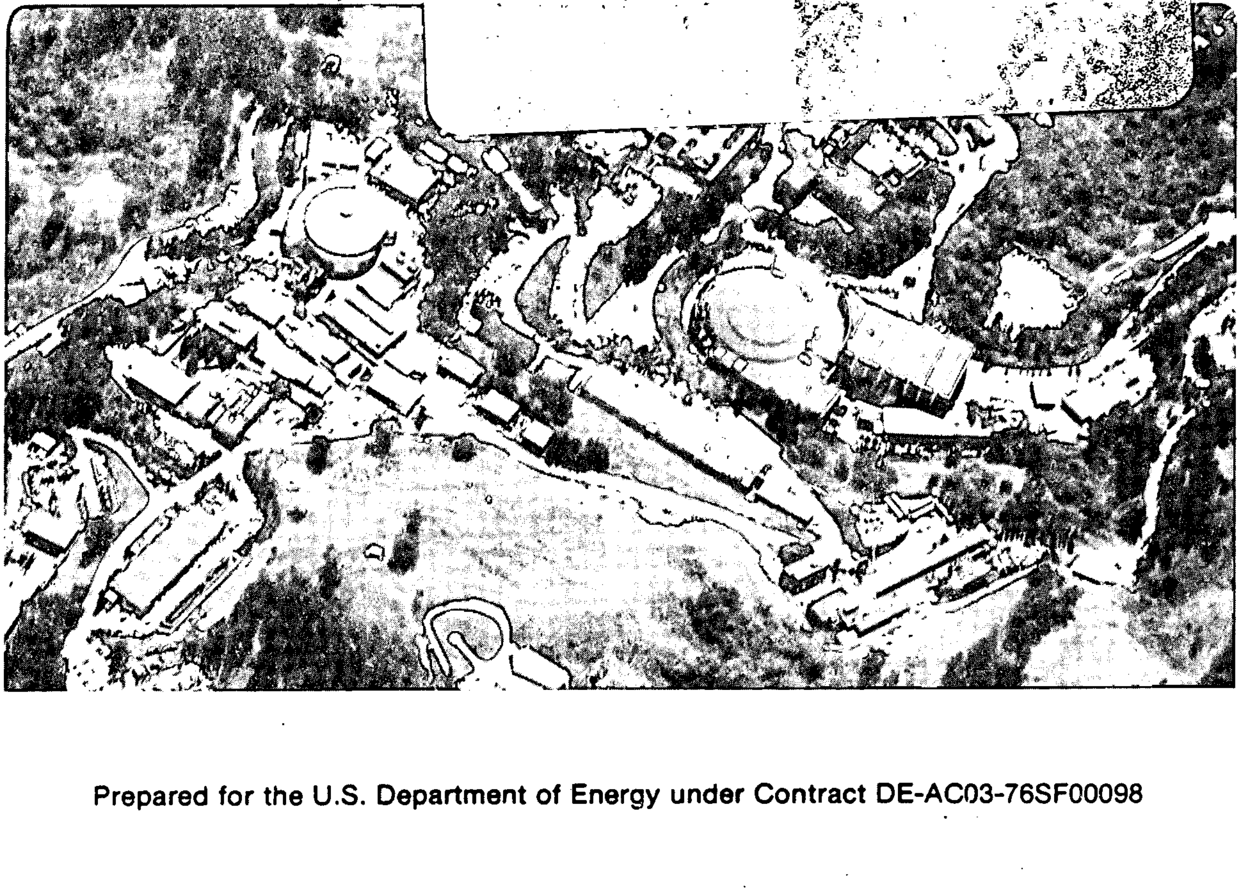
Information and Computing Sciences Division

Submitted to Algorithmica

BIT TRANSPOSITION FOR VERY LARGE SCIENTIFIC AND STATISTICAL DATABASES

H.K.T. Wong, J.Z. Li, F. Olken, D. Rotem, and L. Wong

March 1986



LBL-21269 c.2

Prepared for the U.S. Department of Energy under Contract DE-AC03-76SF00098

LBL-21269

Bit Transposition for Very Large Scientific and Statistical Databases

H.K.T.Wong, J.Z. Li, F.Olken, D.Rotem, and L.Wong

Computer Science Research Department
Lawrence Berkeley Laboratory
University of California
Berkeley, California 94720

March, 1986

This research was supported by the Applied Mathematics Sciences Research Program of the Office of Energy Research, U.S. Department of Energy under contract DE-AC03-76SF00098.

LBL-21269

Bit Transposition for Very Large Scientific and Statistical Databases

Harry K.T. Wong, J. Z. Li*, Frank Olken, Doron Rotem, Linda Wong
Lawrence Berkeley Laboratory,
University of California
Berkeley, California 94720

Abstract

Conventional access methods cannot be effectively used in large Scientific /Statistical Database (SSDB) applications. A file structure (called bit transposed file) is proposed which offers several attractive features that are better suited for the special characteristics that SSDBs exhibit. This file structure is an extreme version of the (attribute) transposed file. The data is stored by vertical bit partitions. The bit patterns of attributes are assigned using one of several data encoding methods. Each of these encoding methods is appropriate for different query types. The bit partitions can also be compressed using a version of the run length encoding scheme. Efficient operators on compressed bit vectors have been developed and form the basis of a query language. Because of the simplicity of the file structure and query language, optimization problems for database design, query evaluation, and common subexpression removal can be formalized and efficient exact solution or near optimal solution can be achieved. In addition to selective power with low overhead for SSDBs, the bit transposed file is also amenable to special parallel hardware. Results from experiments with the file structure suggest that this approach may be a reasonable alternative file structure for large SSDBs.

1. Introduction and Motivation

Scientific/Statistical Databases (SSDBs) exhibit many specialized data usage and characteristics ([12, 15]). Despite the advent of many advanced access methods, the dominant file structure for very large SSDBs is still the simple sequential file. The major reason is a "mismatch" between conventional access methods such as inverted files, B-trees, hashing, etc. and the characteristics of SSDBs. First, since the cardinality of SSDBs attributes is typically small, most access methods simply partition the database into a small number of still very large files, with prohibitively expensive overhead for the pointers, structures, tables, etc., with only limited selective power added. Second, since SSDBs are largely static, the expensive overhead associated with the dynamic facilities of most access methods is not justified. Third, the values of SSDBs attributes tend

Supported by the Office of Energy Research, U.S. DOE under Contract No. DE-AC03-76SF00098.

* on leave from Dept. of Computer Science, Heilongjiang Univ., China.

to cluster, and current access methods often do not take advantage of this opportunity for compression. Fourth, the access to SSDBs is typically long "sweep" i.e., a long sequence of individual records is fetched and a small number of attributes extracted. This kind of range access is not supported well by most access methods.

The search for an appropriate file structure begins with the fourth point mentioned above, which is the motivation for attribute transposed files ([14, 1]). *Conventional files* store the data as a collection of contiguous records, i.e., all the fields for a single record are stored together on a disk page. *Attribute transposed files* store the data as a collection of contiguous attribute columns, i.e., all of the data for a field (attribute) is stored together. *Bit transposed files (BTF)* store the data as a collection of bit columns, i.e., all of the data for single bit position of an attribute encoding is stored together. Thus the file structure we propose can be seen to be an extreme form of the attribute transposed file.

The basic advantage of attribute transposed files is that only those attribute columns which are needed for a query need be retrieved. In many statistical applications only a small fraction of the attributes are needed for a query. Bit transposed files offer three advantages:

(1) Clever data encodings will permit us to retrieve only a fraction of the bit vectors used to encode an attribute in order to perform a selection.

(2) The bit vectors are amenable to data compression via run length encoding, especially if the data records have been sorted.

(3) Selection criteria can be formulated as boolean expressions on the bit vectors, facilitating fast evaluation and specialized hardware.

In summary the bit transposed file system offers an efficient means of performing selections.

2. Overview

The BTF system has three major components: an index encoder, transposed bit vector loader, and a query processor on bit vectors.

The index encoder translates each field in each record in the database into a series of bits based on several encoding schemes. The result is that each record of the database is translated into a bit pattern.

The second component, called the transposer, stores the bit patterns in a transposed manner so that for each bit position of the bit pattern, a file is produced which contains the bit value of that bit position from all the records in the database. The result is n BTFs where n is equal to the number of bit columns that result after encoding. Because values in large statistical databases tend to cluster, we have developed a compression method to compress the BTFs so that long runs of 0's and 1's can be stored more efficiently.

The third component of this file structure is the query processor on BTFs. The processor translates the retrieval requests on the database into a boolean expression on the BTFs. The translation algorithm takes as input the encoding schemes for the attributes and the query type of the query. The generated boolean expression is then subject to a process called *common subexpression removal* which substitutes subexpressions from the query with results from previous equivalent queries that have been saved by the system. The resultant boolean expression is evaluated by using the primitive boolean operators AND, OR, and NOT that can operate directly on compressed BTFs.

In the rest of the paper, section 3 describes the various index encoding schemes with examples. Also, the problem of optimal index encoding assignment is formalized and solution discussed. Section 4 gives details and examples to the transposition of records by bits. In Section 5, query processing algorithms for decoding of queries and common subexpression removal are presented. Section 6 discusses the implementation and experimentation of the file structure and results are listed in another appendix. In section 7, the related work to BTFs is presented. Section 8 reviews the motivation of the paper. Section 9 discusses our current work and concludes the paper.

3. Index Encoding Schemes and Optimization

In this section we will discuss the available index encoding schemes and the problem of optimal index encoding assignment in our BTF transposed file structure. Index encoding schemes are crucial to BTFs because they ultimately decide how many boolean operations have to be performed on the bit vectors. There are four basic schemes: binary, k-of-n, unary, and superimposed. Each one of these schemes can have a composite version for attributes with large number of values. Below we will describe each of them with examples and discuss the usage of the scheme for different kind of queries. In section 3.6, the problem of automating the design of optimal index encoding schemes is discussed and solution presented.

3.1. Binary Encoding

Given an attribute A with n possible values, the binary encoding of A is to use $\log_2(n)$ bits for each value v and the bit pattern for v is the binary number in the range of 0 and n, corresponding to the ordinal integer of v among the n values of A. As a convention, the bit positions are labeled $b_0, b_1, ..., b_m$, from the rightmost bit to the leftmost. This scheme requires the minimum of storage but all bits have to be examined for retrieval.

As an example throughout this paper, we will use an application of radiation experiment on dogs. This experiment database contains information such as dog type, weight, age, dosage, location, etc. Assume that there are 10 dog types. To encode dog type using the binary encoding requires 4 bits and the bit patterns of these 10 values range from 0000 to 1010.

3.2. K-of-N Encoding

This encoding scheme assigns bit patterns to attribute values by turning on a distinct set of K bits from N bits. Hence it can encode up to $\binom{N}{K}$ values. For example, the 1-of-10 encoding for dog type mentioned above would involve the following bit patterns:

```
0000000001
0000000010
0000000100
   ...
1000000000
```

An 2-of-5 encoding for dog type has the following bit patterns:

```
00011
00101
00110
01001
01010
01100
10001
10010
10100
11000
```

Unlike binary encoding, this scheme requires examining only K bits for any value. It allows a time-space tradeoff in the sense that more storage space (larger N) would mean less bits to examine (smaller K).

3.3. Unary Encoding

This scheme requires N bits to encode N values and it is useful for attributes that are involved mostly in range or inequality queries. For example, the following is the result of encoding dog type using the unary encoding scheme.

```
0000000001
0000000011
0000000111
   ...
1111111111
```

To retrieve all dog types that are larger than type 3 requires to examine only bit b_3 (if it is 1 or not). Similarly for all dog types that are below type 3 requires to examine only bit b_2 (if it is 0 or not). Range queries in the form of (a,b) can be expressed as two inequality queries in the form of $<$ a and $>$ b. For example, to find all dog types between 3 and 8 requires examining only bits b_2 (greater than 2) and b_8 (less than 9). Similarly queries such as $\neq$a can be expressed as $<$ a or $>$ a. For example, to find all dog types not equal to dog type 3 requires examining bits b_2 (less than 3) and b_3 (greater than 3).

3.4. Superimposed Encoding

Superimposed encoding scheme ([9]) is important for SSDBs which contain large volume of bibliographical data or property data ([12]). To use superimposed encoding for an attribute, a hashing function is first defined which maps each desired keyword in the attribute into a bit pattern of N bits. Given an attribute value (text with keywords), the collection of bit patterns of all the keywords are superimposed (logically ORed together) and the resulting bit pattern is the encoded value. This scheme supports partial match queries. Given a list of keywords to be searched, the keywords are hashed, superimposed onto a bit vector and the resulting bit pattern is matched against the superimposed codes of the attribute. Because of the possible "false drops", this scheme can only be used as a "filter" in the sense that only some records not qualifying are eliminated but of the selected ones, a search for the keywords is still required to reject those that were selected because their codes coincide with the superimposed code of the query.

3.5. Composite Encoding

Each of the four encoding schemes mentioned above can be made "composite". Given an encoding scheme E and a bit vector with length N, a composite encoding scheme for E of D fields is the concatenation of D groups of bit vectors, each of which is encoded using E and with length N. For example, suppose there are 1000 possible values for the attribute dosage in our experiment database. An 1-of-1000 encoding would require 1000 bits for each value. A composite 1-of-10 encoding with 3 fields, which involves the concatenation of three 1-of-10 fields together, can be used. To find a particular dosage value, only 3 bits have to be examined, 1 from each field. Composite k-of-n encoding with d fields can be viewed as a n-bit radix number with d digits. It is not required for the fields of a composite encoding scheme to have the same length. For the example above, we could have the first field encoded as 2-of-5 and the last two as 1-of-10.

Given an attribute encoded in a particular scheme, to find the correspondence between a value of the attribute and its bit pattern is done by a code table lookup. The major advantage of the composite encoding scheme is the reduction of the code table size. The reason is that the number of possible encoded values of a composite encoding scheme is the *product* of the number of possible encoded values of its fields, but the size of its code table is just the *sum* of the size of the code tables of its fields. In fact, in the case that all fields have the same encoding, then the same code table can be used. Another advantage of composite encoding is that for attributes with large number of possible values, multiple levels of grouping can be made so that selection can be performed based on the disired level. For example, in the composite encoding of dosage above (three 1-of-10 fields), there are three levels of grouping of values, one at the hundreds, one at the tens, and one at the ones level. Selection performed at the hundreds, tens, or ones level involves respectively one, two, or three bits. For large SSDBs, having multiple levels of grouping of values is very important and composite encoding scheme is invaluable.

Table 1 summarizes the properties of the encoding schemes. For each scheme, the possible number of encoded values, the number of bits to examine in case of exact match, inequality, or partial match are given. The formulas are expressed in terms of d (the number of fields, in the case of non-composite encoding, d=1), n (the width of each field), and k (the number of bits to turn on in the case of k-of-n encoding). As can be seen from the table, binary encoding schemes provide the most compact encoding in terms of space, but require the examination of all bit positions for exact or inequality queries. K-of-n encoding schemes require the examination of kd bits for exact match, but are expensive to answer inequality queries. Unary encoding schemes are best for inequality search (requires only d bits), but it is space-expensive. None of the above three encoding schemes can handle partial match queries. Superimposed encoding schemes provide such a capability, but they cannot handle other query types.

	# values	exact match	>	partial match
binary	2^{nd}	nd	nd	N/A
k-of-n	$\binom{n}{k}^d$	kd	nd	N/A
unary	$(n+1)^d$	2d	d	N/A
superimposed	$O(2^{nd})$	N/A	N/A	*

Table 1: The properties of encoding schemes

* depends on code density, typically is $\frac{nd}{2}$.

3.6. Index Encoding Optimization

In this section, we will consider automating the optimal index encoding for one encoding scheme, the k-of-n. The result in this section has been generalized to the other encoding schemes ([11]), but for space reason, the generalization will not be presented in this paper.

Given an attribute A with v possible values, the k-of-x encoding method stores each value as a binary number with x digits. Exactly k digits are 1's and the other x-k are 0's. Clearly we can represent at most $\binom{x}{k}$ (the number of combinations of x objects taken k at a time) different values for the attribute using this method and therefore we have the constraint that $\binom{x}{k}$ must be at least v. To meet this constraint we can choose to increase both x and k, increase only x while keeping k small, or increase only k. In any case k will not exceed $\frac{x}{2}$ since $\binom{x}{k}$ is maximized at either $k=\frac{x}{2}$ or $k=\frac{x-1}{2}$ and we will show that increasing k means more boolean operations to answer a query. On the other hand, a large x means that more storage will be required to store the bit vectors. Hence we have a time space tradeoff problem. In this section we address the following problem: Given a certain amount of space to store the bit vectors, what is the optimal partitioning of this space among m attributes such that the expected query processing time is minimized. A more formal definition of the model and a dynamic programming solution to this problem is now given.

Given a database of N records on m attributes $A_1, A_2, \cdots, A_m$, we would like to store the records as a set of bit vectors. The total number of bits reserved for encoding all attributes is C, so that the total storage requirement is C*N. We assume that attribute A_i has v_i possible values and appears in a query with probability p_i. Our problem is to find for each attribute A_i, a k_i and a x_i such that the values for A_i will be encoded in a k_i-of-x_i encoding. We assume that when a value for attribute A_i is mentioned in a query, the amount of boolean operations required to find the appropriate records will be proportional to k_i because this is the number of columns we have to AND / OR in this case[1]. Therefore, minimizing the expected time to answer a query amounts to minimizing

$$\sum_{i=1}^{m} p_i k_i.$$

The constraints are

$$\sum_{i=1}^{m} x_i \leq C.$$

$$\binom{x_i}{k_i} \geq v_i.$$

We observe that the minimum value for any x_i is $\log_2(v_i)$, by information theoretic arguments and also the maximum value for k_i that we will consider is $\log_2(v_i)$ because otherwise we can use the usual binary encoding with this cost for query processing. The above optimization problem can be solved by dynamic programming techniques by using the following principal of optimality. Let us denote by

[1] To simplify the presentation, we only consider exact match queries here. In [11], a more general problem will be addressed.

$$OPT_y(1,2,\cdots,j)$$

the optimal expected query cost for the above problem where we only consider attributes $A_1,A_2,..A_j$ and allow these attributes to use a total of y bits. We observe that

$$OPT_w(1,2,....,j+1) = \min_y\{OPT_y(1,2,..,j)+OPT_{w-y}(j+1)\}.$$

In words, every partitioning of w bits for the first $j+1$ attributes is achieved by finding some y where $y<w$ such that the first j attributes use y bits and the attribute A_{j+1} uses the remaining $w-y$ bits. Among all such feasible partitionings, we have to find the value for y which minimizes the sum of these costs. This provides us with an iterative approach where at each iteration we add one more attribute into consideration until we finally find $OPT_C(1,2,..,m)$ which is the optimal way of partitioning C bits among m attributes. A program which implements this idea was written in PASCAL and it took a very short time to compute optimal allocations for all practical size databases that we are currently using in our experiments. The details of the testing of the algorithm appear in Appendix A.

4. Bit Transposition

In this section we will describe the file structure using some examples. The steps in obtaining the BTFs involve the following: first, the encoding schemes are decided for selected attributes; then the attributes are encoded for all records in the database; for each bit position of the encoded record, a file consisting of all the bits across the whole database is generated and stored; finally, the files are compressed.

The database of radiation experiment on dogs is used again here to illustrate these steps. The attributes of the database include the dog type, weight, age, dosage, location, observation, etc. Assume the following encoding schemes

attribute	**# values**	**scheme**
dog type	10	2-of-5
weight	8	unary (8 bits)
age	20	binary (5 bits)
dosage	200	composite unary (3 fields of 6 bits)
location	10	1-of-10
observation	1000 keywords	superimposed on 10 bits

Using these encoding schemes, the database is transformed into bit patterns. For each bit position, a bit vector is stored as a file. For the example above, the number of bit vectors files is as follows:

attribute	**#bit vectors**
dog type	5
weight	8
age	5
dosage	18
location	10
observation	10

These bit vectors are then subject to compression. The compression method we use is a variation of the header compression scheme proposed by [3], which in turn is a variation of the run length encoding scheme with efficient access to the compressed data. Because of space limitation, the reader is referred to the above paper for the details of the compression method. The BTF compression scheme has the additional capability of suppressing the compression in the case where the overhead exceeds the gain of compression. This happens when there are a large number of short runs of 1's and 0's. The suppression algorithm involves look-ahead and constant evaluation and balance of the cost of the overhead vs the storage gain from the compression.

5. Query Processing

In this section, the query processing aspects of BTFs are presented. The primitive operators on bit vectors are first discussed, followed by a short description of the query language, then the algorithms for decoding of queries and common subexpression removal.

5.1. Boolean Operators on Bit Vectors

The primitive operators on bit vectors are the boolean operators AND, OR, and NOT. These operators can be efficiently implemented by breaking up the bit vectors into words and feed to the boolean operators of the CPU. More efficiency is gained when the compression rate of the bit vectors is large. In the case of computing the AND operator between two bit vectors, for example, the runs of 0's in one of the bit vectors can be "skipped", and the corresponding part of the other bit vector can also be skipped. For bit vectors with large compression rate (which is one of the dominant characteristics of SSDBs), this skipping action can be used to produce very fast boolean operators over bit vectors.

5.2. Query Language

The current BTF query language is a simple boolean expression language which allows range, exclusion, and set conditions. For example, to retrieve all female dog records between age 3 to 5 and weigh more than 10 lbs, the following query can be used.

$$\text{sex}=1 \wedge \text{age}=3{:}5 \wedge \text{weight}>10$$

The query "retrieve all dogs except German Sheppards (which has value 105) or dogs that have developed cancer in the brain", can be expressed as

$$\text{dogtype}\neq 105 \vee \text{observation}=\{\text{"cancer"},\text{"brain"}\}$$

(Note that in the current implementation of the BTF there is actually a menu-driven user interface which alleviates the user from having to memorize the internal codes of the attributes.)

In general, all queries have the forms:

$$\bigvee_{i=1}^{n}\left(\bigwedge_{k=1}^{m_i}(\text{EXP}_{i_k})\right) \tag{1}$$

where EXP_{i_k} can have the following form: (1) $A\,\theta\,V$, where θ is in $\{<,>,\neq\}$ and V is a value of attribute A; (2) A=VS, where VS is a set of values from attribute A; and (3) $A=V_1{:}V_2$, where V_1 and V_2 are values of attribute A.

5.3. Decoding of queries

Given a query, a series of table lookup has to be performed to translate the query into boolean expression of bit vectors. The first table is the attribute index encoding table which records the encoding scheme for each attribute and contains a pointer to the attribute's bit assignment table. The bit assignment table records the bit pattern for each attribute value. In the case of composite encoding, there can be up to d bit assignment tables where d is the number of fields of the composite encoding scheme.

Given the bit assignments for each attribute in the query, the next step is to generate boolean expression on bit vectors. Below, we will illustrate this step by some examples.

1. Simple exact match queries.

(a) find all German Shepherds

From table lookup, value 105 is found to have bit assignment 01100 in a 2-of-5 encoding scheme. The query

$$\text{dogtype}=105$$

is translated to

$$\text{dogtype}\ (b_3 \wedge b_2).$$

and can now be evaluated. (Remember that the bits are named from right to left.)

(b) find all 5-year-old dogs.

Age 5 is encoded as 00101 in a binary encoding scheme, so the following expression is generated

$$\text{age }(\overline{b_4} \wedge \overline{b_3} \wedge b_2 \wedge \overline{b_1} \wedge b_0)$$

(c) find all 5-year-old German Shepherds.

is translated to

$$\text{dogtype}(b_3 \wedge b_2) \wedge \text{age }(\overline{b_4} \wedge \overline{b_3} \wedge b_2 \wedge \overline{b_1} \wedge b_0).$$

2. Queries with set conditions

find all dogs that have been radiated on locations 1, 4, or 7.

The query is expressed as

$$\text{location}=\{1,4,7\}$$

Since location is encoded as a 1-of-10, the query is translated to

$$\text{location }(b_0 \vee b_3 \vee b_6).$$

3. Queries with range conditions

(a) find all dogs lighter than weight class 7.

Recall that attribute weight is encoded as unary, the above query is translated simply to

$$\text{weight }(\overline{b_6}).$$

(b) find all dogs receiving 30 or more dosage units.

Attribute dosage is encoded as a Composite unary with 3 fields of 6 bits. Assume dosage 30 is encoded as 000111,000011,011111. The query can be translated to

$$\text{dosage }((b_{14} \wedge b_7 \wedge b_4) \vee (b_{14} \wedge b_8) \vee b_{15})$$

In the section to follow, the translation algorithm will be presented more formally. Given a query in form (1), if EXP_{i_k} has been translated into boolean expressions B_{i_k}, we can easily translate the query into

$$\bigvee_{i=1}^{n}(\bigwedge_{k=1}^{m_i} B_{i_k}).$$

Below the translation schemes of expression EXP_{i_k} to B_{i_k} are given, classified by the different encoding schemes. In the following, the symbol "$\rightarrow$" is defined as a short hand notation for "is translated into".

1. Binary Encoding Scheme.

Let attribute A have N bit vectors called $b_1, b_2, \ldots, b_N$, and the value to decode is $V=e_1, e_2, \ldots, e_N$. Among the e's, assume that $e_{i_1}, e_{i_2}, \cdots, e_{i_m}$ are 1. Also, let the value set to decode is $VS=\{V_1, \cdots, V_p\}$, and $V_i=e_{i1}, \cdots, e_{iN}$ for i=1 to p. The translation scheme of binary encoding for the various forms of boolean expressions involving A, VS and V are as follow:

(a) A=VS, denote the result of translation as $B_=(V)$

$$A=VS \rightarrow \bigvee_{i=1}^{p} \bigwedge_{j=1}^{N} X_j^i,$$

where,

$$X_j^i = \begin{cases} b_j & \text{if } e_{ij}=1 \\ \overline{b_j} & \text{if } e_{ij}=0 \end{cases}$$

(b) A>V, denote the translation result to be $B_>(V)$

$$A>V \rightarrow \bigvee_{p=0}^{m}[(\bigwedge_{j=1}^{p} b_{i_j}) \wedge (\bigvee_{j=i_p+1}^{i_{p+1}-1} b_{i_j})],$$

where $i_0=0$ and $i_{m+1}-1=N$,

(c) A<V, denote the translation result as $B_<(V)$

$$A<V \rightarrow \overline{B_=(V) \vee B_>(V)},$$

(d) $A \neq V$,

$$A \neq V \rightarrow \overline{B_=(V)},$$

(e) $A=V_1:V_2$,

$$A=V_1:V_2 \rightarrow B_=(V_1) \bigvee B_=(V_2) \bigvee (B_>(V_1) \bigwedge B_<(V_2)).$$

2. K-of-N Encoding Scheme.

Let the value set VS be the same as before, and each element V_q in VS have the form $e_{q1}, \ldots, e_{qN}$. Among these e's, k of them are 1, and they are denoted by $e_{qi_1}, \cdots, e_{qi_k}$. The translation scheme of K-of-N encoding for A=VS is as follows:

$$A=VS \rightarrow \bigvee_{q=1}^{p} \bigwedge_{j=1}^{K} b_{qi_j}.$$

The other forms of boolean expressions have the same translation schemes as presented in binary encoding above.

3. Unary Encoding scheme.

Let attribute A have N bit vectors called $b_1, \ldots, b_N$, and the value to decode is $V=e_1, \ldots, e_i, e_{i+1}, \ldots, e_N$ where $e_i=0$ and $e_{i+1}=1$. Let the value set to decode be $VS=\{V_1, \ldots, V_p\}$, and $V_j=e_1, \ldots, e_{i_j}, e_{i_j+1}, \ldots, e_N$ where $e_{i_j}=0$ and $e_{i_j+1}=1$ for j=1 to p. The translation schemes are as follows:

(a) A=VS

$$A=VS \rightarrow \bigvee_{j=1}^{p} \overline{b_{i_j}} \bigwedge b_{i_j+1},$$

(b) A>V

$$A>V \rightarrow b_i,$$

(c) A<V

$$A<V \rightarrow \overline{b_{i+1}},$$

(d) $A \neq V$

$$A \neq V \rightarrow b_i \bigvee \overline{b_{i+1}},$$

and $A=V_1:V_2$ is similar to binary encoding scheme.

The translations for superimposed encoding scheme are the same as that for binary encoding. For composite encoding scheme, the translation schemes are the combination of the translation schemes presented above.

5.4. Common Subexpression Removal

The usage patterns of SSDBs often exhibit strong locality of reference in the sense that subsets of the database are often isolated and analysed intensively over a period of time. During this period, the queries against the data will often have a large amount of common subexpressions. The result of evaluating these subexpressions can be saved and used to simplify the future queries, as a result, better performance. This process is referred to as *common subexpression removal* ([4, 5, 6, 7, 10]) in the literature. We have solution for this problem in the context of BTF involving conjunctive queries (the majority of SSDBs queries involve only the AND operator in our experience).

A *temporary result database* is maintained which keeps track of the queries and their corresponding bit vectors as a result of previous query evaluations. When a new query arrives, this database is consulted to determine if substitutions can be made to the query from these saved queries (called subexpressions from this point on) so that fewer boolean operations are required to perform on the bit vectors. In addition, the temporary result database is regularly updated by a policy that observes the Least Recently Used (LRU) practice. That is, if space for the temporary result database runs out, the subexpression (and its corresponding bit vector) which is used least in substitutions is removed in favor of a new query.

Formally, a conjunctive query can be represented as a set where the elements are the primitive conditions on the corresponding attributes. The objective of common subexpression removal is to find the least number of subexpressions (from the temporary result database)) that contain the maximum number of elements of the incoming query. Let

$$S_1=\{a_{11},a_{12},\cdots,a_{1k_1}\}$$
$$S_2=\{a_{21},a_{22},\cdots,a_{2k_2}\}$$
$$\cdots$$
$$S_n=\{a_{n1},a_{n2},\cdots,a_{nk_n}\}$$

be a set of subexpressions that have been saved. Let

$$S=\{a_1,a_2,\cdots,a_m\}$$

be the new query. The problem is to find a collection S's, call it Π

$$\Pi=\{S_{i1},S_{i2},\cdots,S_{ip}\}$$

so that

$$S \cap S_{iq} \neq \phi \text{ and } \bar{S} \cap S_{iq} = \phi \quad \text{for } i \leq q \leq p$$

and

$$||\Pi|| + ||S - \bigcup_{S_i \in \Pi} S_i|| \text{ is minimal.}$$

This problem is similar to the set covering problem [9] except that the query set is not required to be covered entirely and for a subexpression to be eligible as a source of substitution, all the elements in a subexpression must appear in the query set. The set covering problem is a NP-complete problem. Below we will describe an efficient heuristic that has been shown to provide very close approximation to optimal solutions.

Define a subexpression to be a *candidate* if all its elements appear in the query expression. The algorithm G below is a greedy heuristic that accepts an incoming query (referred to as Q in G) and selects candidates in descending order of their sizes. The output of G is a shorter but equivalent query where the elements that the candidates cover are replaced by the candidates themselves.

Algorithm G

```
S ← φ;  /* S collects eligible candidates */
DO WHILE (there exists candidate with size ≥ 2)
      Find the largest candidate, call it lc;
      Remove the elements in lc from Q;
      Save lc in S as part of the solution;
ENDDO
Q ← Q ∪ S;
```

Q contains the final elements to be evaluated. This simple algorithm produces optimal solutions in over 99% of our test queries. Our experiment shows that on the average, the common subexpression removal process reduces the size of incoming queries by about 20%. The formal treatment of the common subexpression removal problem in terms of analysis and detailed experimentation is presented in a separate paper [16].

6. Implementation

A prototype of the BTF structure has been implemented in a VAX/VMS environment using mainly C with some assembler coding. The physical level of the prototype includes a compression package, an index encoder, a bit vector bulk loader, a set of boolean operators on compressed bit vectors. At the logical level, we have an user interface module, and a query processer. The architecture of the system is given in Figure 1 in terms of control and data flow.

The largest database we have running using the bit transposed file is a 110,000 records cancer incidents database available from the National Institute of Health. Some informal performance experiments were performed comparing the retrieval time of the BTF with Datatrieve, a DEC relational DBMS, against the cancer data. The selected queries are typical inquiries on cancer data, according to specialists in our laboratory. The result is that BTF incur much smaller overhead (up to 10 times) and the retrieval time is consistently 10 times or more faster than Datatrieve. More details of some of experiments can be found in Appendix B. Besides the space and retrieval time, the loading time of the data is also of interest. We selected four attributes of the cancer database to have transposed bit vectors. Indices for the same attributes were generated in Datatrieve for a fair comparison. The transposition of the records into bit vectors took about half an hour on our VAX, but it took Datatrieve 5 days to create two indices and 9 days for 4 indices. In fact, only about 75% of the database was loaded because of the excessive CPU time.

7. Related Work

As we mentioned in the Motivation Section, the basis of our approach is the transposed file, which is popular among SSDB implementors ([13]). The BTF can be thought of as an extreme version of the transposed file. In addition to the advantages associated with the transposed file for SSDBs, the bit transposed file offers three potential benefits: indexing capability with minimum of overhead because bit vectors are data *and* indices; better compression rate because of the front compression opportunity (such as a telephone book) and the lack of word, or even byte boundary; and the inherent parallelism (and hence efficiency) associated with the boolean logic on bit vectors.

Two earlier and simple versions of the BTF appear in [2] and [8]. The former only has the binary encoding scheme whereas the latter only the 1-of-n scheme. Neither consider other encoding schemes for different query types, compression of bit vectors, or optimization problems.

Suppose we encode an attribute with large cardinality of values with a 1-of-n encoding, and then apply run length encoding compression to each bit vector. This is equivalent to a fully inverted file with difference encoded inverted lists (for each attribute value). By varying the encoding, we can interpolate (in terms of space and access time) between fully inverted files and simple sequential files. In [1], index encoding techniques are used as a compression method where the cardinality of the value set of an attribute can change with time. The index encoding techniques presented in this paper are used primarily as a means to provide tradeoff and optimization of storage and retrieval speed. Also, since the static nature of SSDBs is emphasized, the dynamic property of index encoding schemes as proposed in [1] is avoided on purpose.

8. Summary

The motivation of our research began with the examination of why current access methods are not in use for large SSDB processing. We will review our observations and examine whether our proposal provides part of the solution.

The first characteristic of SSDBs is that attributes tend to have small cardinality. As a result, most current access methods would add limited selective power yet incur large overhead. The BTF takes advantage of this property because small cardinality of attributes implies that it is possible to have small number of bit vectors, hence values can be efficiently retrieved. Also, there is minimal overhead associated with bit vectors because bit vectors are data *and* indices.

The second characteristic of SSDBs is the clustering effect of attribute values. The BTF takes advantage of this property by compressing the bit vectors. Unlike traditional compressed data, however, there is no need to uncompress in order to use the data. Instead the compressed bit vectors are used to implement efficient boolean operators.

The third characteristic is the static (or append only) property of SSDBs that tend to underuse the dynamic mechanism of most access methods. Transposed files (especially bit transposed files) exhibit poor update performance because they require a disk seek per attribute (bit) vector for each record modified, unless updates are batched. We presently provide only append operations for BTFs.

The fourth characteristic of SSDBs is that queries tend to access many records but only on a few attributes. This property is the basic motivation of the transposed files. The BTF can be thought of as a transposed file with a built-in "generalized" indexing mechanism which incurs minimal overhead. Generalized indices because the elaborate index encoding schemes provide a continuum of indexing levels based on access requirements and storage considerations.

Because of the simplicity of the file structure and query language, optimization problems for database design, query evaluation, and common subexpression removal can be formalized and efficient exact solution or near optimal solution can be achieved.

9. Current work and Conclusion

From our experience of implementing the BTF, it is apparent that simple yet powerful multiprocessor hardware can be built to support the file structure. We have a preliminary design for a transposer and a VLSI design for a boolean logic machine. The transposer consists of a 32 by 32 register matrix. 32 words (32 bits each) are read in at a time and the bits are sliced into the matrix horizontally. The transposition is done by reading the data vertically from the top 32 registers. The entire database can be transposed using this matrix. The same transposer can also be used to convert from the bit transposed form to record format. The boolean logic machine is organized as a tree where each node is a simple processor with only AND, OR, and NOT operations built in. Given a query, the "tree machine" is dynamically reconfigured to correspond to the parse tree of the query. The data, which is in the form of bit vectors, is fed to the tree machine from the leafs. The result is propagated upward in a pipeline manner towards the root, which produces the result. A prototype 8-processor chip has been designed. The processors are connected in a full crossbar which has the necessary logic to make it dynamically reconfigurable.

Another optimization opportunity we are currently exploring is the determination of the optimal order of evaluating the bit vectors in a query to minimize running time. The idea is to take advantage of the different compression rates of the bit vectors. Large compression rate of a bit vector implies that the skipping action by the boolean operators mentioned earlier will be more pronounced, as a result, more efficiency can be gained ([11]).

We envision the BTF to be used in coexistence with other access methods, especially in situations where efficient index encoding is difficult to obtain. Examples include attributes with continuous domains and very large cardinality. Our current implementation of the BTF, in fact, accommodates other file structures such as sequential files, and transposed files. We are also extending the concept of BTFs so that hierarchical relationships can be modelled and manipulated efficiently ([17]).

In conclusion, we believe that the BTF offers an interesting approach to SSDBs because of its simplicity, low overhead, inherent efficiency due to the parallel bit operations in computers, the optimization opportunities, and amenability to parallel hardware implementation.

Acknowledgements

We would like to thank Arie Shoshani for his valuable comments. Credits are to Michael Ger for implementing the index encoding algorithm and providing the test data. We would also like to thank the referees for their helpful comments.

Appendix A Index Encoding Optimization Algorithm Result

This appendix lists the test runs and the CPU time it took the optimization algorithm to obtain the optimal results. Table 2 contains the input and output of the test runs. For each test run, each attribute has two pairs of numbers. The left number of the upper pair represents the number of possible values for the attributes and the right number is the frequency of the attribute being accessed. The lower pair of numbers (a, b) represents the result of the optimal bit assignment.

Fig. 2 shows the CPU time comparison of the exhaustive search method and our dynamic programming method. In some instances, the latter's running time is less than 1% of the brute force method. As can be seen, this method is efficient enough for most practical databases.

Appendix B Performance Comparison

The database is a real cancer incidents records. It contains information such as the patient's sex, age, cancer site, type of cancer cells, year, etc.

Table 3 lists the size of the test database in Datatrieve and BTF. The overhead column of BTF is the size (in number of 512-byte pages) of the bit vectors. The overhead for Datatrieve is the size of the indices.

The list of queries contains twenty queries, ten in BTF syntax, and ten in Datatrieve syntax.

Fig. 3 shows comparison of running time of the listed queries by the BTF system and Datatrieve.

List of Queries

1. B: year=75
   D: find r01key4 with year = 75

2. B: year=73:78
   D: find r01key4 with year bt 73 and 78

3. B: year=73:77 $\wedge$ racerea=2
   D: find r01key4 with year bt 73 and 77 and racere = 2

4. B: year={75,77} $\wedge$ sexre=1
   D: find r01key4 with (year = 75,77) and (sexre = 1)

5. B: sexre=1 $\wedge$ racerea=1
   D: find r01key4 with sexre = 1 and racere = 1

6. B: year=74 $\wedge$ agere=10:12
   D: find r01key4 with year = 74 and agere bt 10 and 12

7. B: site=570:579 $\wedge$ sexre=1
   D: find r01key4 with site bt 570 and 579 and sexre = 1

8. B: year=76:78 $\wedge$ sexre=2
   D: find r01key4 with (year bt 76 and 78) and sexre = 2

9. B: year={73,75,77} $\wedge$ site=859
   D: find r01key4 with year = 73, 75, 77 and site = 859

10. B: year={76,78} $\vee$ histolog={9730,9731}
   D: find r01key4 with (year = 76,78) or (site = 9730, 9731)

References

1. Batory, D.S., "Index Encoding: A Compresion Technique for Large Statistical Databases", Proc. 2nd Workshop on Statistical Database Management, 1983, pp 306-314.

2. Batory, D.S., "On Searching Transposed Files," *ACM Transaction on Database Systems,* Vol. 4, no. 4, Dec., 1979, 531-544.

3. Brill, R.C, Tolken, S.E., "Subset Selection by Boolean Calculation", Unpublished memo, 1977.

4. Eggers, S., Olken, F., Shoshani, A., "A Compression Technique for Large Statistical Databases", in Proc. 1981 International Conference on Very Large Data Bases, Cannes, France, Sept, 1981.

5. Finkelstein, S., "Common Expression Analysis in Database Applications", Proc. of 1982 ACM SIGMOD Conference, Boston, MA, pp. 235-245.

6. Jarke, M., "Common Subexpression isolation in Multiple Query Optimization", in *Query Processing in Database Systems,* Springer-Verlag, 1985.

7. Kambayashi, Y., Ghosh, S., "Query Processing Using the Consecutive Retrieval Property", in *Query Processing in Database Systems,* Springer-Verlag, 1985.

8. Kim, W., "Global Optimization of Relational Queries: A First Step", in *Query Processing in Database Systems,* Springer-Verlag, 1985.

9. Kiyoki, Y., Tanaka, K., and Aiso, H., "Design and Evaluation of a Relational Data Base Machine Employing Advanced Data Structures and Algorithms", in The 8th Annual Symposium on Computer Architecture, May 12-14, 1981, Minneapolis, Minn.

10. Knuth, D.E., *The Art of Computer Programming,* Volume 3, Addison Wesley, 1973.

11. Larson, P.A., Yang, H.Z., "Computing Queries from Derived Relations", Proc. of 1985 International Conference on Very Large Data Bases, Stockholm, Sweden.

12. Li, J.Z., Wong, H.K.T., "On Formal Properties of Bit Transposed Files", 1986, LBL Technical Report LBL-21281.

13. Shoshani, A., Olken, F., Wong, H.K.T., "Characteristics of Scientific

Databases", Proc. 1984 International Conference on Very Large Data Bases, Singapore, 1984.

14. Turner, M., Hammond, R., Cotton, F., "A DBMS for Large Statistical Databases," Proc. 1979 International Conference on Very Large Data Bases, Rio de Janeiro, 1979.

15. Wiederhold, G., *Database Design,* McGraw-Hill, 2nd Edition, 1983.

16. Wong, H.K.T., "Micro/Macro Statistical/Scientific Database Management", The First IEEE International Conference on Data Engineering, Los Angeles, March, 1984.

17. Wong, H.K.T., Li, J.Z., Rotem, D., "Common Subexpression Removal for Bit Transposed Files", 1986, LBL Technical Report LBL-21283.

18. Wong, H.K.T., Li, J.Z., "Hierarchical Bit Transposed Files", 1986, LBL Technical Report LBL-21284.

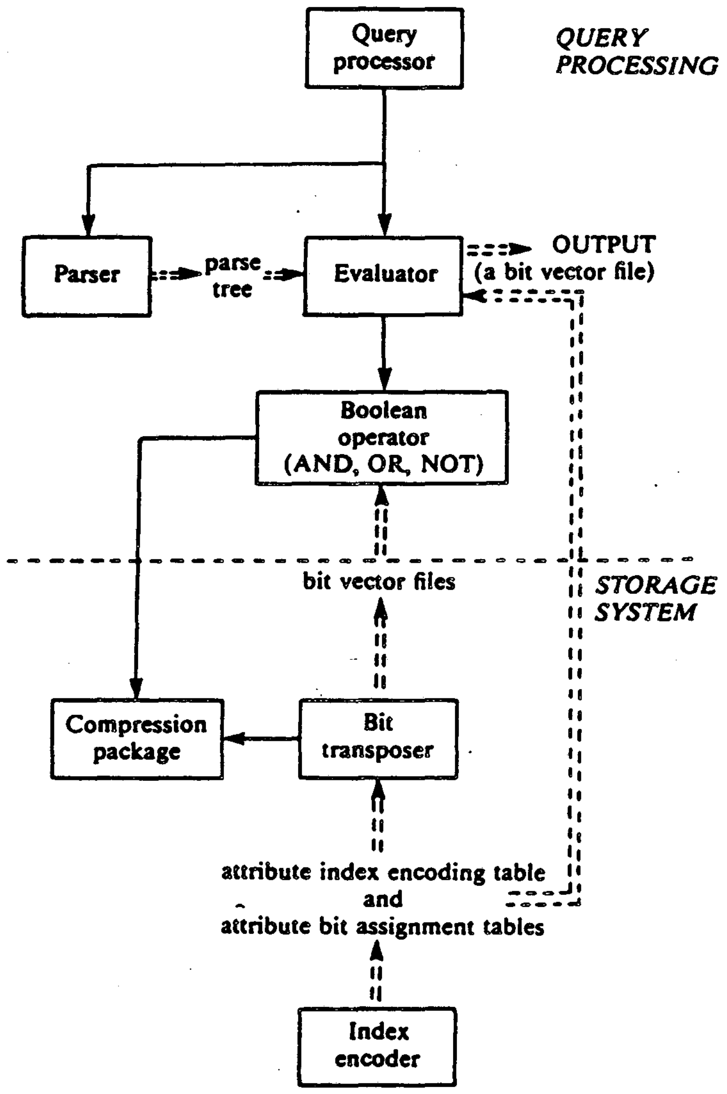


Fig. 1 Architectural view of BTF (→, functional flow; ⇒, output-input).

Table 2

Run	Max. bits	Attr. 1	Attr. 2	Attr. 3	Attr. 4	Attr. 5	Attr. 6	Attr. 7	Attr. 8	Attr. 9	Attr. 10
1	70	89000, 30 (19, 9)	2567, 20 (14, 6)	780, 30 (12, 5)	1000, 2 (10, 16)	5, 6000 (5, 1)	40, 60 (10, 2)				
2	80	800, 20 (10, 10)	56, 400 (6, 3)	70, 3000 (13, 2)	667, 30 (10, 10)	20, 400 (6, 3)	769, 90 (12, 5)	2, 1000 (1, 1)	36, 600 (10, 2)	50, 300 (6, 3)	4, 200 (2, 2)
3	70	5670, 30 (15, 7)	456, 20 (9, 9)	900, 70 (13, 5)	690, 200 (14, 4)	456, 30 (9, 9)	690, 20 (10, 10)				
4	80	9400, 30 (14, 14)	600, 60 (12, 5)	56, 400 (8, 3)	70, 20 (7, 7)	700, 1000 (13, 4)	60, 600 (12, 2)	9, 100 (4, 4)	567, 30 (10, 10)		
5	80	6790, 30 (13, 13)	69000, 200 (23, 6)	34567, 90 (19, 7)	23, 1000 (5, 7)	560, 20 (10, 10)	90, 30 (7, 7)				
6	80	500, 20 (13, 4)	600, 30 (13, 4)	700, 10 (12, 5)	60, 100 (12, 2)	30, 200 (9, 2)	6, 100 (6, 1)	36, 9 (6, 6)	25, 10 (5, 5)	46, 100 (11, 2)	3, 1000 (3, 1)

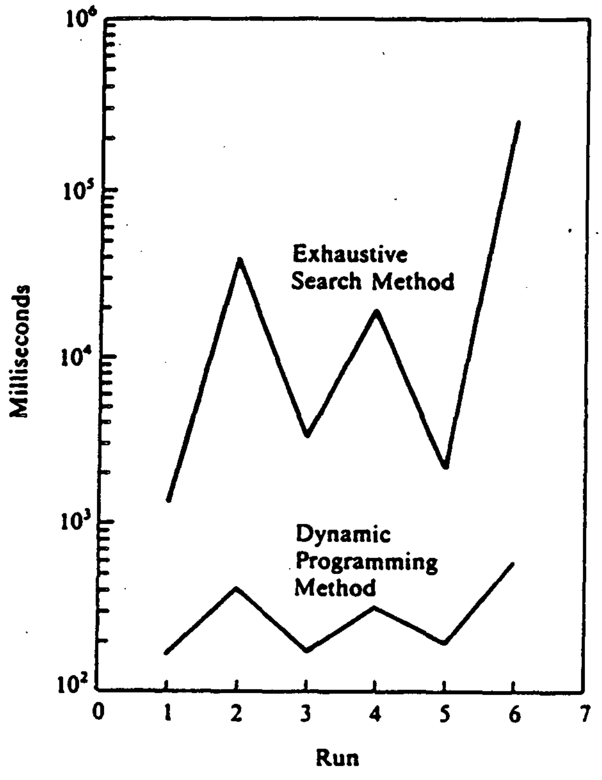


Fig. 2

Table 3

	Number of records	Database size (in pages)	Overheads (in pages)
BTF	110,000	6,974*	1,332
Datatrieve	83,729†	8,100	10,134

* The size of the database after four attributes are index encoded.
† Only about 75% of the original datbase is loaded because of excessive CPU time.

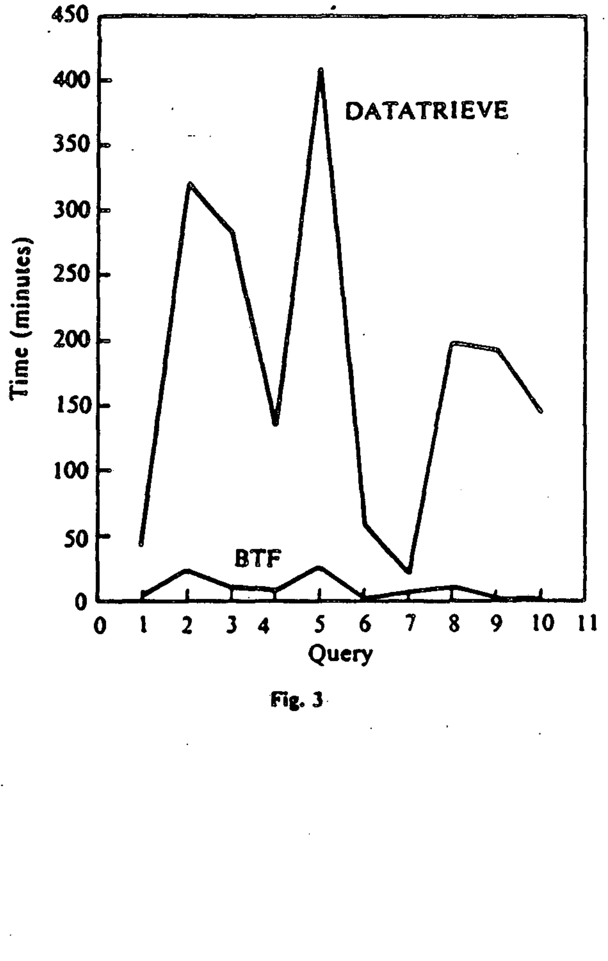


Fig. 3