# STORAGE STRUCTURES AND ACCESS METHODS IN THE RELATIONAL DATA BASE MANAGEMENT SYSTEM INGRES

Gerald Held and Michael Stonebraker
Department of Electrical Engineering and Computer Sciences
and the Electronics Research Laboratory
University of California, Berkeley, California 94720
(415) 642-4871

INGRES is a relational data base management system, under development at Berkeley, which implements a high level non-procedural query language (QUEL). We describe our choice of storage structures which are used to implement relations. Also described is an access method interface which provides a single relational view of all storage structures.

## I. INTRODUCTION

INGRES (INteractive Graphics and REtrieval System) is a relational data base management system which is being implemented on a PDP-11/40 based hardware configuration at Berkeley. INGRES runs as a user job on top of the UNIX operating system [RITC73] developed at Bell Telephone Laboratories, and the implementation of INGRES is programmed in "C"[RITC74], a high level language in which UNIX itself is written.

In order to implement a high level, non-procedural query language it was convenient to define a low level, procedural language for data access. Such a language has been implemented and provides a relational view of data to higher level software while supporting a variety of actual storage structures for efficient data access. We will describe this access method interface language and discuss the five storage structures which it currently supports. The discussion will center on the considerations made in choosing the set of storage structures.

In order to motivate the discussion we first give some examples of interactions allowed in the query language, QUEL, and then indicate the mechanism used to decompose interactions into access method calls.

## II. QUEL

QUEL (QUEry Language) has points in common with Data Language/ALPHA [CODD71], SQUARE [BOYC73] and SEQUEL [CHAM74] in that it is a complete [CODD72] query language which frees the programmer from concern for how data structures are implemented and what algorithms are operating on stored data. As such it facilitates a considerable degree of data independence [STON74b]. Since basic entities in QUEL are relations, we define them and indicate the sample relation which is used in the examples in this paper.

Given sets $D_1,\ldots,D_N$ (not necessarily distinct) a RELATION $R(D_1,\ldots,D_N)$ is a subset of the Cartesian product $D_1 \times \ldots \times D_N$. In other words, R is a collection of N-tuples $X = (X_1,\ldots,X_N)$ where $X_i$ is an element of $D_i$ for i in $\{1,\ldots,N\}$. The sets $D_1,\ldots,D_N$ are called DOMAINS of R and R has DEGREE N. The only restriction put on relations in QUEL is that they be normalized. Hence, every domain must be SIMPLE, i.e. it cannot have members which are themselves relations.

Clearly, R can be thought of as a table with elements of R appearing as rows and with columns labeled by domain names as illustrated by the following example.

| | NAME | DEPT | SALARY | MANAGER | AGE |
|---|---|---|---|---|---|
| EMPLOYEE | Smith | toy | 10000 | Jones | 25 |
| | Jones | toy | 15000 | Johnson | 32 |
| | Adams | candy | 12000 | Baker | 36 |
| | Johnson | toy | 14000 | Harding | 29 |
| | Baker | admin | 20000 | Harding | 47 |
| | Harding | admin | 40000 | none | 58 |

The above indicates an EMPLOYEE relation with domains NAME, DEPT, SALARY, MANAGER and AGE. Each employee has a manager (except for Harding, who is presumably the company president), a salary, an age and is in a department.

Each column in a tabular representation for R can be thought of as a function mapping R into $D_i$. These functions will be called ATTRIBUTES. An attribute will not be separately designated but will be identified by the domain defining it.

A QUEL interaction includes at least one RANGE statement of the form:

```
RANGE OF variable-list IS relation-name
```

The symbols declared in the range statement are variables which will be used as arguments for tuples. These are called TUPLE VARIABLES. The purpose of this statement is to specify the relation over which each variable ranges.

Moreover, an interaction includes one or more statements of the form:

```
Command Result-name (Target-list)
        WHERE Qualification
```

Here, Command is either RETRIEVE, APPEND, REPLACE, or DELETE. For RETRIEVE and APPEND, Result-name is the name of the relation which qualifying tuples will be retrieved into or appended to. For REPLACE and DELETE, Result-name is the name of a tuple variable which, through the qualification, identifies tuples to be modified or deleted. The Target-list is a list of the form

```
Result-domain = Function,...
```

Here, the Result-domain's are domain names in the result relation which are to be assigned the value of the corresponding function.

The following suggest valid QUEL interactions. A complete description of the language is presented in [HELD75a].

Example 2.1 Find the birth date of employee Jones

```
RANGE OF E IS EMPLOYEE
RETRIEVE INTO W(BDATE = 1975 - E.AGE)
WHERE E. NAME = 'JONES'
```

Here, E is a tuple variable which ranges over the EMPLOYEE relation and all tuples in that relation are found which satisfy the qualification E.NAME = 'JONES.' The result of the query is a new relation, W, which has a single attribute, BDATE, that has been calculated for each qualifying tuple. If the result relation is omitted, qualifying tuples are printed on the user's terminal. Also, in the Target-list, the 'Result-domain =' may be omitted if Function is of the form Variable.Attribute (i.e. NAME = E.NAME may be written as E.NAME - see example 2.6).

Example 2.2 Insert the tuple (Jackson,candy,13000, Baker,30) into EMPLOYEE.

```
APPEND TO EMPLOYEE(NAME = 'Jackson', DEPT = 'candy',
   SALARY = 13000, MGR = 'Baker', AGE = 30)
```

Here, the result relation EMPLOYEE is modified by adding the indicated tuple to the relation.

Example 2.3 Delete the information about employee Jackson.

```
RANGE OF E IS EMPLOYEE
DELETE E WHERE E.NAME = 'Jackson'
```

Here, the tuples corresponding to all employees named Jackson are deleted from EMPLOYEE.

Example 2.4 Give a 10 percent raise to Jones

```
RANGE OF E IS EMPLOYEE
REPLACE E(SALARY BY 1.1 * E.SALARY)
WHERE E.NAME = 'Jones'
```

Here, E.SALARY is to be replaced by 1.1*E.SALARY for those tuples in EMPLOYEE where E.NAME = 'Jones.' (Note that the keywords IS and BY may be used interchangeably with '=' in any QUEL statement.)

Also, QUEL contains aggregation operators including COUNT, SUM, MAX, MIN, AVG and the set operator SET. Two examples of the use of aggregation follow.

Example 2.5 Replace the salary of all toy department employees by the average toy department salary.

```
RANGE OF E IS EMPLOYEE
REPLACE E(SALARY BY AVG(E.SALARY WHERE E.DEPT =
      'toy')) WHERE E.DEPT = 'toy'
```

Here, AVG is to be taken of the salary attribute for those tuples satisfying the qualification E.DEPT = 'toy.' Note that AVG(E.SALARY WHERE E.DEPT = 'toy') is scalar valued and consequently will be called an AGGREGATE. More general aggregations are possible as suggested by the following example

Example 2.6 Find those departments whose average salary exceeds the company-wide average salary, both averages to be taken only for those employees whose salary exceeds $10000.

```
RANGE OF E IS EMPLOYEE
RETRIEVE INTO HIGHPAY(E.DEPT)
WHERE AVG(E.SALARY BY E.DEPT WHERE E.SALARY > 10000)
   > AVG(E.SALARY WHERE E.SALARY > 10000)
```

Here, AVG(E.SALARY BY E.DEPT WHERE E.SALARY>10000) is an AGGREGATE FUNCTION and takes a value for each value of E.DEPT. This value is the aggregate AVG(E.SALARY WHERE E.SALARY>10000 AND E.DEPT = value). The qualification expression for the statement is then true for departments for which this aggregate function exceeds the aggregate AVG(E.SALARY WHERE E.SALARY>10000).

## III. DECOMPOSITION

The basic mechanism of processing statements in QUEL now follows. All update statements are processed into one or more RETRIEVE statements followed by a sequence of calls to the access methods to insert, delete or modify tuples. A RETRIEVE statement with more than one tuple variable is decomposed into a sequence of RETRIEVE statements each with a single tuple variable as described in [HELD75a]. The mechanism used is one of "tuple substitution." Here, we describe the algorithm for aggregate free interactions.

Consider a query involving one or more tuple variables $X = (X_1,\ldots,X_N)$ with a range $R_1 \times \ldots \times R_N$. Denote the qualification by Q(X) and suppose Q(X) is expanded into conjunctive normal form so that it consists of clauses connected by AND with each clause containing atomic formulas connected by OR. An atomic formula can contain only the boolean operator NOT.

Algorithm

1. stop if query has only a single variable

2. For each variable, say $X_1$ with Range $R_1$, collect all attributes which depend on $X_1$ and all clauses in the qualification which depend only on $X_1$. Say $D_1,\ldots,D_k$ are the attributes and the clauses put together yield $Q_1(X_1)$.

Issue the query:

RANGE OF $X_1$ IS $R_1$<br>
RETRIEVE INTO $R_1'(X_1.D_1,\ldots,X_1.D_k)$<br>
WHERE $Q_1(X_1)$

3. Replace the range $R_1$ in the original query by $R_1'$. The purpose of 2. and 3. is to limit each tuple variable to as small a relation as possible before continuing to step 4.

4. Take the variable with the fewest tuples in its range and substitute in turn the values of its tuples. This reduced the number of variables by one. After each substitution repeat 1.-3.

Step 4 is called tuple substitution and represents the most time-consuming step for multivariable queries. The choice of which variable to substitute for is critical. Our criterion (the one with the fewest tuple variables) is by no means optimal in general.

In this manner a multivariable query is reduced to a sequence of one-variable queries and calls to the access methods to obtain tuples for substitution. A one variable query is interpreted by a "one-variable query processor" (OVQP). This processes accesses tuples from the indicated relation one at a time, checks if the qualification is true for that tuple and if so assembles the target-list attributes and inserts them into the result relation. Besides interpreting the qualification and target-list, this processor must:

1. ascertain if any secondary indices [STON74c] can profitably be used to speed access.

2. attempt to restrict the number of tuples accessed to less than all tuples in the relation.

To accomplish 1 and 2, it requires the help of the access method routine FIND. This command and the rest of the access method interface are described in the next section.

## IV. ACCESS METHOD INTERFACE

To find all the tuples in a relation which satisfy the indicated qualification, the OVQP must either access and test all tuples in the relation, or else must determine that only a subset of the relation need be tested with knowledge that the remainder of the relation cannot satisfy the qualification. One way the OVQP might make such a determination is indicated in the following example. If the EMPLOYEE relation is sorted on increasing values of the SALARY domain, then in processing the query:

```
RANGE OF E IS EMPLOYEE
RETRIEVE E.NAME
WHERE E.SALARY < 10000 AND E.MGR = 'Jones'
```

the OVQP can stop testing tuples as soon as a tuple is encountered which has the SALARY domain greater than 10000. Depending on the particular storage structure which is in use for a given relation and the domains specified in the qualification, the OVQP may or may not be able to limit the number of tuples examined. Instead of having the OVQP and higher level software be concerned with this problem, a relational access method interface language (AMI) has been implemented. This language frees higher level software from details of actual storage structures and thus allows restructuring of relations for more efficient operation as interaction conditions change. It has points in common with Gamma Zero [BJOR73] and XRM [LORI74]. Relation access and update through AMI is accomplished in the following manner.

1. A scan a relation is begun by using the FIND statement to supply any information in the qualification which might be of help in limiting the range of the scan. FIND examines the information provided, and in conjunction with a knowledge of the storage structure used to implement the relation, determines starting and ending points for the scan.
2. Beginning from the starting point tuples are accessed, one at a time, using the GET statement until the ending point is reached where GET returns an end of scan condition. The programmer may not assume that the tuples will be returned in any particular order.
3. Each tuple has a unique identifier called the tuple id (TID) which is returned with the tuple. This tuple id may be used to refer back to the tuple for re-access or updating (usually done after the qualification has been tested for the tuple).
4. INSERT, DELETE, and REPLACE statements are supported for all storage structures and respectively, add one new tuple to a relation, remove one tuple, or change the value of an existing tuple. When using REPLACE or DELETE the user must supply a TID to indicate which tuple is to be affected.
5. Apart from scan retrieval, GET also supports direct retrieval of tuples given a TID. This function is used in supporting secondary indices. Briefly, a secondary index is useful for limiting the number of tuples accessed in cases where a value for the primary domain (i.e. the domain used for ordering, for example SALARY above) is not present in the qualification. A secondary index is a relation which has one or more domains from the original relation along with a pointer domain which is an identifier of a tuple in the indexed relation. For instance, if SALARY is the ordering domain in EMPLOYEE, it might be useful to have a secondary index on NAME. To build the secondary index all that is needed is a query of the form:

   ```
   RANGE OF E IS EMPLOYEE
   RETRIEVE INTO NAMEINDEX(E.NAME,PTR=E.TID)
   ```

   This relation may then be stored in a structure which has NAME as the primary (ordering) domain. When a query on the EMPLOYEE relation specifies a value for NAME, the OVQP may access tuples in the NAMEINDEX relation and use the domain PTR as a TID to be supplied to GET which will return the corresponding tuple in the EMPLOYEE relation. Although two relations must be used to access tuples, a costly scan of the whole EMPLOYEE relation may be avoidable.

For AMI to support a new storage structure, the following must be done.

1. A correspondence must be defined between a TID and a physical position in the structure.
2. There must be a linear ordering defined on TID's so that successive calls to GET will return all tuples in the relation.
3. FIND, GET, REPLACE, DELETE and INSERT functions must be implemented for the new structure.

## V. CHOICE OF STORAGE STRUCTURES

We will now consider the types of storage structures currently supported in INGRES. First we divide all storage structures into two classes, keyed structures and non-keyed structures. A keyed structure is one in which a domain (or combination of domains) of a tuple is used to determine where in secondary storage the tuple should be stored. In such structures, when a value of the key domain is specified, the tuple(s) having the specified value can be located directly without a full scan of the relation. In a non-keyed structure, however, a full scan of the relation is always required.

Non-keyed structures are supported in INGRES in the form of unsorted tables. These structures are used as the initial form of all relations resulting from user queries. They are also useful in moving data to and from standard UNIX files. Non-keyed structures may have secondary indices to provide faster access; however, relations which are used frequently are normally converted to some form of keyed structure in order to provide more efficient access on the most frequently used key (domain).

To begin the discussion of keyed structures, we define several terms which will be used in the remainder of the paper.

K key space - a set of possible values for the key domain. The key domain (or primary key) is the domain of the relation which is used in determining the storage location for tuples in the relation. Several domains may be combined to form a single key. However for the discussion we will assume only one domain is used. The key space will be taken to be the interval (a,b) of the real line since other data encodings may be transformed to this set.

F key distribution - a usually unknown probability distribution function which describes how the keys are distributed over the range (a,b).

N the number of tuples in the relation.

$\{K_1,\ldots,K_N\}$ the N keys, $K_i \in K$, present in the relation. This is a sample from F. To simplify notation, we will assume the sample has been ordered so that $K_i < K_{i+1}$ for all i.

A address space - a set of integers {1,2,...,R}, each member of the set representing a secondary storage location (page) capable of storing one or more tuples. These R pages are referred to as the "primary pages." When a primary page becomes full, one or more "overflow pages" are linked to it.

R the number of primary pages.

C tuple capacity of a page - the number of tuples that can be accomodated on a single page of secondary storage (page size divided by tuple size).

C' key capacity of a page - the number of keys that can be accomodated on a single page of secondary storage (page size divided by key domain size).

H key to address function - a mapping from key values to addresses $H: K \rightarrow A$.

P parameter set - a set of parameters which are used in the key to address transformation, H.

OCF occupancy factor - a measure of secondary storage space usage. It is defined as the total secondary storage space used (primary plus overflow pages) divided by minimum possible space (the minimum space is N/C).

ACF access factor - average number of data page accesses to reach a tuple. This includes the primary data page access and all overflow page accesses, but does not include any accesses required by the key to address transformation.

The differences between types of keyed structures lie in the definitions of the key to address function, H. Two desirable conditions for this function to meet are:

Condition 1.

> The function should not introduce additional secondary accesses in order to compute an address.

Condition 2.

> The function should map the given sample of the key space uniformly across the address space.

Condition 1 implies that the function should have as few parameters as possible. Condition 2 requires that pages in secondary storage are used in a uniform manner so that overflow areas are not heavily used. Overflow areas are necessary when more than C tuples are mapped to a single address. To access tuples on an overflow page, first the primary page (the one determined by H) must be accessed and then the overflow page(s) is accessed. The added accesses necessary to retrieve tuples on overflow pages increases ACF.

## VI. RANDOMIZING FUNCTIONS

A class of functions which usually meet both of these conditions is known as randomizing or hash functions. Here, H is chosen so as to spread the keys randomly across the address space. These functions have the advantage that they meet both conditions 1 and 2 for a large class of key distributions, F. An excellent compilation of various randomizing functions is given in [LUM71a]. Randomizing functions provide an excellent response to the needs of queries involving equality on the key domain. For a given key value, the function H will return the address which contains all tuples possessing that key value. For example, if the EMPLOYEE relation were randomized with SALARY as the key domain, then the query

RETRIEVE (Target-list) WHERE E.SALARY = 10000

would only require an average of ACF accesses to find all qualifying tuples. For these reasons, we have implemented a randomizing structure (using a folding and division scheme).

QUEL, however, provides the ability for queries involving ranges on domain values. For example,

RETRIEVE (Target-list) WHERE E.SALARY > 10000

Since randomizing functions usually have the property that there is no correspondence between the order of the keys and the order of the addresses to which they are assigned, these functions are of little value in selecting tuples from a range of key values. For this reason, a third condition often must be imposed on the key to address function.

Condition 3.

> The function should be an order preserving function (i.e. if $K_1 < K_2$ the $H(K_1) < H(K_2)$).

This condition is important whenever it is expected that queries will involve qualifications which specify a range on the primary key. In such cases it is important to limit the number of tuples scanned to those in the specified range.

We know of no storage structure that satisfies all three conditions independent of the data stored. We now discuss order preserving computed functions (which usually satisfy conditions 1 and 3 but not 2). Then we discuss directory structures (which usually obey conditions 2 and 3 but not 1). Lastly we discuss generalized directories (which offer a continuum of possibilities between the previous two types).

## VII. ORDER PRESERVING COMPUTED FUNCTIONS

This class of functions requires only minimal parameters as in the case of randomizing functions, yet also preserves order in the address space. An example from this class of functions is to take the j leftmost bits of the key as the address [RIVE74]. The value of j is chosen in order to give an address space of $2^j$ values. Another simple, order preserving function is one which divides the key range (a,b) into equal size buckets and assign one of the R address values to each bucket. Here H is defined as

$H(k) = \lceil ((k-a)/(b-a))*R \rceil$ where $\lceil x \rceil$ denotes least integer greater than x.

The advantage of such functions is that they satisfy condition 1 and thus do not introduce any significant overhead in computation of addresses. The problem with all functions of this type is that the distribution in address space is directly dependent on the distribution in key space. So unless there is uniformity in the sample key values, there will be bunching in address space which implies many overflows and/or much wasted primary space. Therefore, we conclude that order preserving computed functions should be used only when it can be determined that "reasonable" uniformity exists in the key space.

## VIII. DIRECTORY STRUCTURES

A normal directory structure is a function which is constructed such that each page contains the same number of tuples and there are initially no overflow pages used. One such function is

$$P = \{L_i | L_i = K_{C*i}, i=1,N/C\}$$

with

$$H(k) = i \text{ for } L_i \leq k < L_{i+1}$$

Here, the parameters of the function are the low key values on each page of secondary storage. This function satisfies condition 2; however it has N/C parameters which means that for nontrivial values of N, the parameters must be stored in secondary memory (violating condition 1). For large values of N, the parameters themselves will need to be located via a key to address function, thus creating the common multilevel directory structure (i.e. ISAM [IBM66]). Each level of the directory adds an additional access to the cost of computing a tuple address. The average access time for a tuple is then the directory access time plus the single data page access (here ACF is 1)

$$\text{Log}_{C'} N/C + 1$$

Despite the cost of directory accesses, this structure is currently widely used when ordering is required. One reason for this choice is that for directories the worst case access time is logarithmic (to a large base) in N, whereas order preserving computed functions may be linear in N.

## IX. GENERALIZED DIRECTORIES

We now combine the two previous ideas into a structure which meets conditions 2 and 3 and has fewer parameters than normal directories. The parameters of a "generalized directory," H, are an ordered set of pairs:

$$P = \{(L_i,A_i) | L_i \in K,\ A_i \in A,\ L_i < L_{i+1},\ A_i < A_{i+1},\ i = 1,M\}$$

The key to address mapping, H, is:

$$H(k) = A_i + \lceil (A_{i+1}-A_i)(k-L_i)/(L_{i+1}-L_i) \rceil$$

$$\text{for } L_i \leq k < L_{i+1}$$

This type of function divides the key space into M intervals which may be of varying sizes and assign to the ith interval $A_{i+1} - A_i$ pages of secondary storage. Within an interval an order preserving computed function is used to divide the key range equally into the pages assigned to that region. Functions of this nature have been investigated by [FEHR75].

A "data independent directory" is one in which the choice of H is made without any knowledge of the distribution of keys within the interval (a,b). An example of such a directory is the order preserving computed function described above where

$M = 2$

$(L_1,A_1) = (a,1)$

$(L_2,A_2) = (b,R)$

A "data dependent directory" is one in which the sample $\{K_1,\ldots,K_N\}$ from the unknown distribution, F, of keys and is used during construction of H. One example of a data dependent directory is the normal directory discussed above where the $L_i$ are chosen to be the low keys on each secondary storage page, i.e.

$$P = \{(L_i,A_i) | L_i = K_{C*i}\ ,\ A_i = A_{i-1} + 1,\ i = 1,\ N/C\}$$

We define a best generalized directory to one which satisfies the following optimization problem.

given a sample $\{K_1,\ldots,K_N\}$ of keys

choose H (as defined above)

with minimum average access time

$\text{Log}_{C'}M + \text{ACF}$

subject to the constraint that the total storage space is less than some factor, $f_1$, greater than the minimum possible storage requirement (N/C pages) i.e. $\text{OCF} < f_1$

The solution to this problem will provide a directory which has the best average access time for the given limitation on total storage space. This optimization attempts to minimize the size, M, of the directory while keeping the overflows to a minimum and remaining within the storage constraints.

Usually the performance of the two limiting cases of generalized directories, order preserving computed functions and pure directories, will not be optimal. In the case of data independent directories, the directory may not be a close approximation of the actual distribution, F, or the initial sample $\{K_1,\ldots,K_N\}$. Therefore, H may map more than C tuples to many addresses requiring the use of excessive overflow pages. On the other hand, normal directories provide an even distribution of keys over address space; however, they require a large number of entries in the directory. Thus average access time in the normal directory may be larger than necessary because of the need to make several accesses to compute the address.

An optimal solution to the above problem would require a prohibitive amount of computation due to the number of degrees of freedom. We therefore redefine the problem as that of finding a minimum directory size (minimum M) for fixed limits on the access factor (ACF) and the occupancy factor (OCF). In this way the inclusion of C' as a parameter is avoided. Even a best solution to this problem would require many passes over the data file (sample keys), so we now outline an algorithm which provides a solution to the second problem with a single pass over the data. Hopefully, this is a good approximation to the first optimization problem.

In the following description, we will refer to the "step width" of the directory function. By this we mean the size of the interval in key space which is mapped to a single value (page) in address space for a given interval of the function (i.e. for the interval from $L_i$ to $L_{i+1}$ the step width is $(L_{i+1}-L_i)/(A_{i+1}-A_i)$.

The algorithm scans the data keys once from lowest key value to highest. At the beginning of the scan, several guesses are made at the step width of the function. As data keys are read, the performance of each of the guesses is measured by computing the access factor and the occupancy factor which would result if that guess were used. When a point is reached in reading data keys where none of the guesses continues to meet the fixed limits, $f_1$ and $f_2$, on occupancy and access factors,

$$\text{OCF} < f_1 \qquad \text{and ACF} < f_2$$

then the point just before the last guess fails is taken as the next entry in the directory. This guess is taken as the step width between the previous entry and the new one. A new set of guesses is then made and the algorithm repeats as above until the last data key is read.

Some comments on the guesses:

1. As a result of the large difference between I/O speeds and computation speeds there is enough CPU time available during a scan of the relation to allow a sizable number of guesses (NGUESS) to be made and tested.
2. If the first key to be scanned in the new interval is $K_i$, then one of the guesses is chosen to be $K_{i+C} - K_i$ (i.e. a step width for which the first page is exactly filled). In the wprst case, this guess will be chosen and will meet the constraints for one page of data keys, resulting in a normal directory structure.
3. By choosing the guesses to be $(K_{i+C*2^j}-K_i)/2^j$ for j = 1, NGUESS we choose points logarithmically distant from the current point and thus get approximations to the slope of the function of both a local and global nature.

The following point should also be carefully noted. As extra space is made available to the algorithm (by increasing the limit on OCF), the algorithm produces a smaller and smaller directory. This is the opposite of what happens in a normal directory which increases in size as extra space is provided.

Some Experimental Results

In Fig. 1 are indicated the results of testing the algorithm on two data files - one is a set of 10,000 uniformly distributed 8 digit numbers between 0 and 99999999; the other data set is a list of 10,000 names of property owners in Alameda County, California. The distribution of names, as would be expected, is quite non-uniform. It is seen that when near uniformity exists in the data space, the algorithm is able to generate a mapping function which uses only 1 percent of the normal directory size if an increase of 30 percent in storage space is tolerable. With non-uniform data such dramatic differences do not appear. However, in this experiment, with a 30 percent increase in space, a large enough savings is made in directory size to save a level in the directory (from 3 to 2) and thus decrease average access time substantially.

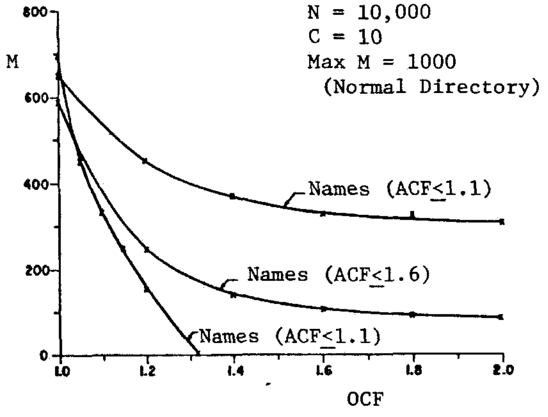

Figure 1

We are implementing this general directory structure in INGRES and will monitor its performance on a wide variety of data. Such a structure will take advantage of whatever uniformity exists in data sets and will never give worse performance than a normal directory.

## X. STATIC vs. DYNAMIC DIRECTORIES

In the above discussion we have only been concerned with the process of building key to address functions and using them for data retrieval. We now consider the problem of choosing a structure which will be useful in an environment which includes updates to the relation (REPLACE, DELETE, and APPEND). We consider two different approaches to the problem of maintaining directory structures in this environment.

### Static Directory Structures

Here, a directory is built as described above and is not altered during updates. Insertions are handled by chaining tuples into overflow areas after space in primary pages is full. When overflow areas become full, a reorganization is required which rebuilds the directory. An example of this structure is [IBM66].

### Dynamic Directory Structure

In this scheme a directory is built as above. However, inserts are handled by splitting a data page when it overflows into two pages and enlarging the directory to point to both pages. If this enlargement causes the directory page to split, the process iterates to the next level up and can continue all the way back to the root in the worst case. In this way the tree is kept balanced, eliminating the need for periodic reorganizations. An example of this structure is B-trees [BAYE70]. Here, the number of pages that each directory page can point to varies between k + 1 and 2k+1 for a given k. This k is determined by the page size and key size of the file.

There are variations of both types of directories. For example, B*-trees are discussed in [KNUT73] and offer obvious advantages over B-trees. The original proposal suggested that whole tuples be put in directory pages. In fact, placing only keys in the directory increases k and thereby makes the tree have fewer levels. Hence data pages can be accessed with fewer retrievals from secondary storage. VSAM [KEEH74] is another variant of dynamic directories.

We feel dynamic directory structure suffer from the following serious flaws

a) problems with concurrency

Suppose two processes are simultaneously accessing a dynamic directory; one inserting a tuple and one performing a scan over a portion of the tree. Suppose further that the scanning process is part way through a page when the updating process causes that page to be split by the insertion. This rearrangement will leave the scanning process pointing to the wrong (or a nonexistent tuple) unless the updating process alters the scan pointer of all other processes in a nontrivial way.

Other problems also arise when two processes concurrently update the same B*-tree. Suppose that the two processes are adding a tuple to two adjacent pages in the tree and suppose both pages are full. Each process must lock the page it is updating since it will be altered. Then it must examine the two adjacent pages to see if tuples can be spilled over to them and a split avoided. However, the adjacent page is locked and each process is requesting access to the page the other has locked. Clearly, this deadlock situation must be recognized and broken.

b) problems with secondary indices

Suppose a dynamic directory is constructed on one key and a secondary index is desired on another key or combination of keys. It is reasonable for the index to be a second dynamic directory with a pointer to a tuple in the first tree as a data item. In this case every time the primary directory is rearranged by splitting (and thereby causing tuples to be moved), the pointers in the secondary index must also be updated for all tuples which have been moved. To avoid this latter update, the data items in the index directory must be keys in the first tree and not pointers. If so, reference to a tuple in the primary directory by utilizing the index requires a search of both dynamic directories. Hence, retrieval is slower than if pointers could be used.

c) problems with fan out

Because pages are split on the fly in a dynamic directory, explicit pointers to data pages must be present in the higher levels of a dynamic directory. These pointers consume space and limit the value of k that can be attained.

These problem are all avoided in static directory structures. A static directory structure can have the property that tuples are never moved; thus pointers can be safely used in secondary indices. Moreover, the directory is static; therefore, an updating process need only ever lock the page it is modifying and no others. Also, since tuples are not moved, there is no danger of a scanning process pointing to a non-existent tuple. Lastly, pointers in the directory can be easily suppressed thus increasing the fanout possible (often by as much as a factor of two). Of course, the primary disadvantage of a static directory structure is the necessity of periodic reorganization when the overflow area becomes highly utilized.

For the above reasons we are implementing a static directory structure and are monitoring activity in the file and automatically reorganizing it (by rewriting the directory and moving all tuples back into the primary area) when necessary.

## XI. COMPRESSED STORAGE STRUCTURES

In addition to these structures, we have implemented one of the many possible page compression schemes for both randomized and generalized directory structures. In these two structures, data is stored in a more highly coded form which requires less secondary storage space at a cost of somewhat higher computational time for decoding and encoding during retrieval and update.

## XII. SUMMARY

We have described a part of the implementation considerations in the relational data base system, INGRES. A description was given of the access method interface language and the storage structures supported. These structures are:

1. Non-keyed structures
2. Randomized keyed structures
3. Generalized directories
4. Compressed randomized keyed structures
5. Compressed generalized directories

The structures are used for data relations and for secondary indices on data relations where desired. Among the storage structures is a more general directory structure which should provide better performance than normal directory structures. Currently, the access methods have been implemented for all of the storage structures discussed and a monitoring system is being designed to automatically choose the most desirable structure depending on interaction conditions. A future report will discuss the strategy used in this monitor.

ACKNOWLEDGEMENT

Research sponsored by the National Science Foundation Grant GK-43024x, U.S. Army Research Office--Durham Contract DAHC04-74-G0087, the Naval Electronic Systems Command Contract N00039-75-C-0034, a Grant from the Sloan Foundation and an RCA David Sarnoff Fellowship.

## REFERENCES

[BAYE70] Bayer, R. and McCreight, E., "Organization and Maintenance of Large Ordered Indices," Proc. 1970 ACM-SIGFIDET Workshop on Data Description, Access and Control, Houston Texas, Nov. 1970.

[BJOR73] Bjorner, D., Codd, E. F., Deckert, I. L., and Traiger, I. L., "The Gamma Zero n-ary Relational Data Base Interface: Specifications of Objects and Operations," IBM San Jose Research Report RJ1200, Apr. 1973.

[BOYC73] Boyce, R., et al., "Specifying Queries as Relational Expressions: SQUARE," IBM Research, San Jose, Ca., RJ1291, Oct. 1973.

[CHAM74] Chamberlin, D. and Boyce, R., "SEQUEL: A Structured English Query Language," Proc. 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974.

[CODD71] Codd, E. F., "A Data Base Sublanguage Founded on the Relational Calculus," Proc. 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, Ca., Nov. 1971.

[CODD72] Codd, E. F., "Relational Completeness of Data Base Sublanguages," Courant Computer Science Symposium 6, May 1972.

[FEHR75] Fehr, E. S., "A Cost Study of Directory Structures for Ordered Files," Master's Thesis, University of Texas at Austin, Jan. 1975.

[HELD75a] Held, G. D., Stonebraker, M. and Wong, E., "INGRES - A Relational Data Base Management System," Proc. 1975 NCC, AFIPS Press, 1975.

[IBM66] "OS ISAM Logic," IBM, White Plains, N.Y., GY28-6618.

[KEEH74] Keehn, D. G. and Lacy, J. D., "VSAM Data Set Design Parameters," IBM Systems Journal, Vol. 13, No. 3, pp. 186-213, 1974

[KNUT73] Knuth, D., The Art of Computer Programming Vol. 3, Addison-Wesley, Reading, Mass. 1973.

[LORI74] Lorie, R. A., "XRM - An Extended (n-ary) Relational Memory," IBM Cambridge Scientific Center Tech. Rep. 320-2096, Jan. 1974.

[LUM71a] Lum, V. Y., Yuen, P. S. T. and Dodd, M., "Key-to-Address Transform Techniques: A Fundamental Performance Study on Large Existing Formatted Files," CACM Vol. 14, No. 4, pp. 228-239, Apr. 1971.

[RITC73] Ritchie, D. and Thompson, K., "The UNIX Time Sharing System," CACM Vol. 17, No. 7, pp. 365-375, July 1974.

[RITC74] Ritchie, D. M., "C Reference Manual," UNIX Programmer's Manual, Bell Telephone Laboratory, Murray Hill, N.J., July 1974.

[RIVE74] Rivest, R. L., "Analysis of Associative Retrieval Algorithms," IRIA Report No. 54, Feb. 1974.

[STON74b] Stonebraker, M., "A Functional View of Data Independence," Proc. 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974.

[STON74c] Stonebraker, M., "The Choice of Partial Inversions and Combined Indices," International Journal of Computer and Information Sciences, Vol. 3, No. 2, 1974.