The Design and Implementation of INGRES

MICHAEL STONEBRAKER, EUGENE WONG, AND PETER KREPS
University of California, Berkeley
and
GERALD HELD
Tandem Computers, Inc.

The currently operational (March 1976) version of the INGRES database management system is described. This multiuser system gives a relational view of data, supports two high level nonprocedural data sublanguages, and runs as a collection of user processes on top of the UNIX operating system for Digital Equipment Corporation PDP 11/40, 11/45, and 11/70 computers. Emphasis is on the design decisions and tradeoffs related to (1) structuring the system into processes, (2) embedding one command language in a general purpose programming language, (3) the algorithms implemented to process interactions, (4) the access methods implemented, (5) the concurrency and recovery control currently provided, and (6) the data structures used for system catalogs and the role of the database administrator.

Also discussed are (1) support for integrity constraints (which is only partly operational), (2) the not yet supported features concerning views and protection, and (3) future plans concerning the system.

Key Words and Phrases: relational database, nonprocedural language, query language, data sublanguage, data organization, query decomposition, database optimization, data integrity, protection, concurrency
CR Categories: 3.50, 3.70, 4.22, 4.33, 4.34

1. INTRODUCTION

INGRES (Interactive Graphics and Retrieval System) is a relational database system which is implemented on top of the UNIX operating system developed at Bell Telephone Laboratories [22] for Digital Equipment Corporation PDP 11/40, 11/45, and 11/70 computer systems. The implementation of INGRES is primarily programmed in C, a high level language in which UNIX itself is written. Parsing is done with the assistance of YACC, a compiler-compiler available on UNIX [19].

This research was sponsored by Army Research Office Grant DAHCO4-74-G0087, the Naval Electronic Systems Command Contract N00039-76-C-0022, the Joint Services Electronics Program Contract F44620-71-C-0087, National Science Foundation Grants DCR75-03839 and ENG74-06651-A01, and a grant from the Sloan Foundation.
Authors' addresses: M. Stonebraker and E. Wong, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Berkeley, CA 94720; P. Kreps, Department of Computer Science and Applied Mathematics, Building 50B, Lawrence Berkeley Laboratories, University of California, Berkeley, Berkeley, CA 94720; G. Held, Tandem Computers, Inc., Cupertino, CA 95014.

The advantages of a relational model for database management systems have been extensively discussed in the literature [7, 10, 11] and hardly require further elaboration. In choosing the relational model, we were particularly motivated by (a) the high degree of data independence that such a model affords, and (b) the possibility of providing a high level and entirely procedure free facility for data definition, retrieval, update, access control, support of views, and integrity verification.

1.1 Aspects Described in This Paper

In this paper we describe the design decisions made in INGRES. In particular we stress the design and implementation of: (a) the system process structure (see Section 2 for a discussion of this UNIX notion); (b) the embedding of all INGRES commands in the general purpose programming language C; (c) the access methods implemented; (d) the catalog structure and the role of the database administrator; (e) support for views, protection, and integrity constraints; (f) the decomposition procedure implemented; (g) implementation of updates and consistency of secondary indices; (h) recovery and concurrency control.

In Section 1.2 we briefly describe the primary query language supported, QUEL, and the utility commands accepted by the current system. The second user interface, CUPID, is a graphics oriented, casual user language which is also operational [20, 21] but not discussed in this paper. In Section 1.3 we describe the EQUEL (Embedded QUEL) precompiler, which allows the substitution of a user supplied C program for the "front end" process. This precompiler has the effect of embedding all of INGRES in the general purpose programming language C. In Section 1.4 a few comments on QUEL and EQUEL are given.

In Section 2 we describe the relevant factors in the UNIX environment which have affected our design decisions. Moreover, we indicate the structure of the four processes into which INGRES is divided and the reasoning behind the choices implemented.

In Section 3 we indicate the catalog (system) relations which exist and the role of the database administrator with respect to all relations in a database. The implemented access methods, their calling conventions, and, where appropriate, the actual layout of data pages in secondary storage are also presented.

Sections 4, 5, and 6 discuss respectively the various functions of each of the three "core" processes in the system. Also discussed are the design and implementation strategy of each process. Finally, Section 7 draws conclusions, suggests future extensions, and indicates the nature of the current applications run on INGRES.

Except where noted to the contrary, this paper describes the INGRES system operational in March 1976.

1.2 QUEL and the Other INGRES Utility Commands

QUEL (QUEry Language) has points in common with Data Language/ALPHA [8], SQUARE [3], and SEQUEL [4] in that it is a complete query language which frees the programmer from concern for how data structures are implemented and what algorithms are operating on stored data [9]. As such it facilitates a considerable degree of data independence [24].

The QUEL examples in this section all concern the following relations.

```
EMPLOYEE  (NAME, DEPT, SALARY, MANAGER, AGE)
DEPT      (DEPT, FLOOR#)
```

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

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

The purpose of this statement is to specify the relation over which each variable ranges. The variable-list portion of a RANGE statement declares variables which will be used as arguments for tuples. These are called *tuple variables.*

An interaction also 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 = QUEL Function. . . .
```

Here the result-domains are domain names in the result relation which are to be assigned the values of the corresponding functions.

The following suggest valid QUEL interactions. A complete description of the language is presented in [15].

Example 1.1. Compute salary divided by age-18 for employee Jones.

```
RANGE OF E IS EMPLOYEE
RETRIEVE INTO W
(COMP = E.SALARY/(E.AGE-18))
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 domain COMP that has been calculated for each qualifying tuple.

If the result relation is omitted, qualifying tuples are written in display format on the user's terminal or returned to a calling program.

Example 1.2. 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. Domains which are not specified default to zero for numeric domains and null for character strings. A shortcoming of the current implemenation is that 0 is not distinguished from "no value" for numeric domains.

Example 1.3. Fire everybody on the first floor.

```
RANGE OF E IS EMPLOYEE
RANGE OF D IS DEPT
DELETE E WHERE  E.DEPT = D.DEPT
            AND D.FLOOR# = 1
```

Here E specifies that the EMPLOYEE relation is to be modified. All tuples are to be removed which have a value for DEPT which is the same as some department on the first floor.

Example 1.4. Give a 10-percent raise to Jones if he works on the first floor.

```
RANGE OF E IS EMPLOYEE
RANGE OF D IS DEPT
REPLACE E(SALARY = 1.1*E.SALARY)
WHERE  E.NAME = "Jones"  AND
       E.DEPT = D.DEPT AND D.FLOOR# = 1
```

Here E.SALARY is to be replaced by 1.1*E.SALARY for those tuples in EMPLOYEE where the qualification is true.

In addition to the above QUEL commands, INGRES supports a variety of utility commands. These utility commands can be classified into seven major categories.

(a) Invocation of INGRES:

```
INGRES data-base-name
```

This command executed from UNIX "logs in" a user to a given database. (A database is simply a named collection of relations with a given database administrator who has powers not available to ordinary users.) Thereafter the user may issue all other commands (except those executed directly from UNIX) within the environment of the invoked database.

(b) Creation and destruction of databases:

```
CREATEDB data-base-name

DESTROYDB data-base-name
```

These two commands are called from UNIX. The invoker of CREATEDB must be authorized to create databases (in a manner to be described presently), and he automatically becomes the database administrator. DESTROYDB successfully destroys a database only if invoked by the database administrator.

(c) Creation and destruction of relations:

```
CREATE relname(domain-name IS format, domain-name IS format, . . .)

DESTROY relname
```

These commands create and destroy relations within the current database. The invoker of the CREATE command becomes the "owner" of the relation created. A user may only destroy a relation that he owns. The current formats accepted by INGRES are 1-, 2-, and 4-byte integers, 4- and 8-byte floating point numbers, and 1- to 255-byte fixed length ASCII character strings.

(d) Bulk copy of data:

```
COPY relname(domain-name IS format, domain-name IS format, . . . ) direction "file-
   name"

PRINT relname
```

The command COPY transfers an entire relation to or from a UNIX file whose name is "filename." Direction is either TO or FROM. The format for each domain is a description of how it appears (or is to appear) in the UNIX file. The relation relname must exist and have domain names identical to the ones appearing in the COPY command. However, the formats need not agree and COPY will automatically convert data types. Support is also provided for dummy and variable length fields in a UNIX file.

PRINT copies a relation onto the user's terminal, formatting it as a report. In this sense it is stylized version of COPY.

(e) Storage structure modification:

```
MODIFY relname TO storage-structure ON (key1, key2, . . . )
INDEX ON relname IS indexname(key1, key2, . . . )
```

The MODIFY command changes the storage structure of a relation from one access method to another. The five access methods currently supported are discussed in Section 3. The indicated keys are domains in relname which are concatenated left to right to form a combined key which is used in the organization of tuples in all but one of the access methods. Only the owner of a relation may modify its storage structure.

INDEX creates a secondary index for a relation. It has domains of key1, key2, . . . , pointer. The domain "pointer" is the unique identifier of a tuple in the indexed relation having the given values for key1, key2, ... An index named AGEINDEX for EMPLOYEE might be the following binary relation (assuming that there are six tuples in EMPLOYEE with appropriate names and ages).

	Age	Pointer
	25	identifier for Smith's tuple
	32	identifier for Jones's tuple
AGEINDEX	36	identifier for Adams's tuple
	29	identifier for Johnson's tuple
	47	identifier for Baker's tuple
	58	identifier for Harding's tuple

The relation indexname is in turn treated and accessed just like any other relation, except it is automatically updated when the relation it indexes is updated. Naturally, only the owner of a relation may create and destroy secondary indexes for it.

(f) Consistency and integrity control:

```
INTEGRITY CONSTRAINT is qualification
INTEGRITY CONSTRAINT LIST relname
INTEGRITY CONSTRAINT OFF relname
INTEGRITY CONSTRAINT OFF (integer, . . . , integer)
RESTORE data-base-name
```

The first four commands support the insertion, listing, deletion, and selective deletion of integrity constraints which are to be enforced for all interactions with a relation. The mechanism for handling this enforcement is discussed in Section 4. The last command restores a database to a consistent state after a system crash.

It must be executed from UNIX, and its operation is discussed in Section 6. The RESTORE command is only available to the database administrator.

(g) Miscellaneous:

```
HELP [relname or manual-section]

SAVE relname UNTIL expiration-date

PURGE data-base-name
```

HELP provides information about the system or the database invoked. When called with an optional argument which is a command name, HELP returns the appropriate page from the INGRES reference manual [31]. When called with a relation name as an argument, it returns all information about that relation. With no argument at all, it returns information about all relations in the current database.

SAVE is the mechanism by which a user can declare his intention to keep a relation until a specified time. PURGE is a UNIX command which can be invoked by a database administratoı to delete all relations whose "expiration-dates" have passed. This should be done when space in a database is exhausted. (The database administrator can also remove any relations from his database using the DESTROY command, regardless of who their owners are.)

Two comments should be noted at this time.

(a) The system currently accepts the language specified as QUEL_1 in [15]; extension is in progress to accept QUEL_n. (b) The system currently does not accept views or protection statements. Although the algorithms have been specified [25, 27], they are not yet operational. For this reason no syntax for these statements is given in this section; however the subject is discussed further in Section 4.

1.3 EQUEL

Although QUEL alone provides the flexibility for many data management requirements, there are applications which require a customized user interface in place of the QUEL language. For this as well as other reasons, it is often useful to have the flexibility of a general purpose programming language in addition to the database facilities of QUEL. To this end, a new language, EQUEL (Embedded QUEL), which consists of QUEL embedded in the general purpose programming language C, has been implemented.

In the design of EQUEL the following goals were set: (a) The new language must have the full capabilities of both C and QUEL. (b) The C program should have the capability for processing each tuple individually, thereby satisfying the qualification in a RETRIEVE statement. (This is the "piped" return facility described in Data Language/ALPHA [8].)

With these goals in mind, EQUEL was defined as follows:

(a) Any C language statement is a valid EQUEL statement.
(b) Any QUEL statement (or INGRES utility command) is a valid EQUEL statement as long as it is prefixed by two number signs (##).
(c) C program variables may be used anywhere in QUEL statements except as

command names. The declaration statements of C variables used in this manner must also be prefixed by double number signs.

(d) RETRIEVE statements without a result relation have the form

```
RETRIEVE (target-list)
         [WHERE qualification]
##{
C-block
##}
```

which results in the C-block being executed once for each qualifying tuple.

Two short examples illustrate EQUEL syntax.

Example 1.5. The following program implements a small front end to INGRES which performs only one query. It reads in the name of an employee and prints out the employee's salary in a suitable format. It continues to do this as long as there are names to be read in. The functions READ and PRINT have the obvious meaning.

```
main( )
{
## char EMPNAME[20];
## int SAL;
while (READ(EMPNAME))
         {
##       RANGE OF X IS EMP
##       RETRIEVE (SAL = X.SALARY)
##       WHERE X.NAME = EMPNAME
                  ##{
                  PRINT("The salary of", EMPNAME, "is", SAL);
                  ##}
         }
}
```

In this example the C variable *EMPNAME* is used in the qualification of the QUEL statement, and for each qualifying tuple the C variable *SAL* is set to the appropriate value and then the PRINT statement is executed.

Example 1.6. Read in a relation name and two domain names. Then for each of a collection of values which the second domain is to assume, do some processing on all values which the first domain assumes. (We assume the function PROCESS exists and has the obvious meaning.) A more elaborate version of this program could serve as a simple report generator.

```
main( )
{
## int VALUE;
## char RELNAME[13], DOMNAME[13], DOMVAL[80];
## char DOMNAME 2[13];
READ(RELNAME);
READ(DOMNAME);
READ(DOMNAME 2);
## RANGE OF X IS RELNAME
while (READ(DOMVAL))
         {
```

```
##        RETRIEVE (VALUE = X.DOMNAME)
##           WHERE X.DOMNAME 2 = DOMVAL
                ##{
                PROCESS(VALUE);
                ##}
          }
}
```

Any RANGE declaration (in this case the one for X) is assumed by INGRES to hold until redefined. Hence only one RANGE statement is required, regardless of the number of times the RETRIEVE statement is executed. Note clearly that anything except the name of an INGRES command can be a C variable. In the above example *RELNAME* is a C variable used as a relation name, while *DOMNAME* and *DOMNAME* 2 are used as domain names.

1.4 Comments on QUEL and EQUEL

In this section a few remarks are made indicating differences between QUEL and EQUEL and selected other proposed data sublanguages and embedded data sublanguages.

QUEL borrows much from Data Language/ALPHA. The primary differences are: (a) Arithmetic is provided in QUEL; Data Language/ALPHA suggests reliance on a host language for this feature. (b) No quantifiers are present in QUEL. This results in a consistent semantic interpretation of the language in terms of functions on the crossproduct of the relations declared in the RANGE statements. Hence, QUEL is considered by its designers to be a language based on functions and not on a first order predicate calculus. (c) More powerful aggregation capabilities are provided in QUEL.

The latest version of SEQUEL [2] has grown rather close to QUEL. The reader is directed to Example 1(b) of [2], which suggests a variant of the QUEL syntax. The main differences between QUEL and SEQUEL appear to be: (a) SEQUEL allows statements with no tuple variables when possible using a block oriented notation. (b) The aggregation facilities of SEQUEL appear to be different from those defined in QUEL.

System R [2] contains a proposed interface between SEQUEL and PL/1 or other host language. This interface differs substantially from EQUEL and contains explicit cursors and variable binding. Both notions are implicit in EQUEL. The interested reader should contrast the two different approaches to providing an embedded data sublanguage.

2. THE INGRES PROCESS STRUCTURE

INGRES can be invoked in two ways: First, it can be directly invoked from UNIX by executing INGRES database-name; second, it can be invoked by executing a program written using the EQUEL precompiler. We discuss each in turn and then comment briefly on why two mechanisms exist. Before proceeding, however, a few details concerning UNIX must be introduced.

2.1 The UNIX Environment

Two points concerning UNIX are worthy of mention in this section.

(a) The UNIX file system. UNIX supports a tree structured file system similar to that of MULTICS. Each file is either a directory (containing references to descendant files in the file system) or a data file. Each file is divided physically into 512-byte blocks (pages). In response to a read request, UNIX moves one or more pages from secondary memory to UNIX core buffers and then returns to the user the actual byte string desired. If the same page is referenced again (by the same or another user) while it is still in a core buffer, no disk I/O takes place.

It is important to note that UNIX pages data from the file system into and out of system buffers using a "least recently used" replacement algorithm. In this way the entire file system is managed as a large virtual store.

The INGRES designers believe that a database system should appear as a user job to UNIX. (Otherwise, the system would operate on a nonstandard UNIX and become less portable.) Moreover the designers believe that UNIX should manage the system buffers for the mix of jobs being run. Consequently, INGRES contains no facilities to do its own memory management.

(b) The UNIN process structure. A process in UNIX is an address space (64K bytes or less on an 11/40, 128K bytes or less on an 11/45 or 11/70) which is associated with a user-id and is the unit of work scheduled by the UNIX scheduler. Processes may "fork" subprocesses; consequently a parent process can be the root of a process subtree. Furthermore, a process can request that UNIX execute a file in a descendant process. Such processes may communicate with each other via an interprocess communication facility called "pipes." A pipe may be declared as a one direction communication link which is written into by one process and read by a second one. UNIX maintains synchronization of pipes so no messages are lost. Each process has a "standard input device" and a "standard output device." These are usually the user's terminal, but may be redirected by the user to be files, pipes to other processes, or other devices.

Last, UNIX provides a facility for processes executing reentrant code to share procedure segments if possible. INGRES takes advantage of this facility so the core space overhead of multiple concurrent users is only that required by data segments.

2.2 Invocation from UNIX

Issuing INGRES as a UNIX command causes the process structure shown in Figure 1 to be created. In this section the functions in the four processes will be indicated. The justification of this particular structure is given in Section 2.4.

Process 1 is an interactive terminal monitor which allows the user to formulate, print, edit, and execute collections of INGRES commands. It maintains a workspace with which the user interacts until he is satisfied with his interaction. The contents of this workspace are passed down pipe A as a string of ASCII characters when execution is desired. The set of commands accepted by the current terminal monitor is indicated in [31].

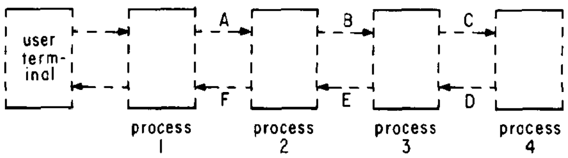

Fig. 1. INGRES process structure

As noted above, UNIX allows a user to alter the standard input and output devices for his processes when executing a command. As a result the invoker of INGRES may direct the terminal monitor to take input from a user file (in which case he runs a "canned" collection of interactions) and direct output to another device (such as the line printer) or file.

Process 2 contains a lexical analyzer, a parser, query modification routines for integrity control (and, in the future, support of views and protection), and concurrency control. Because of size constraints, however, the integrity control routines are not in the currently released system. When process 2 finishes, it passes a string of tokens to process 3 through pipe B. Process 2 is discussed in Section 4.

Process 3 accepts this token string and contains execution routines for the commands RETRIEVE, REPLACE, DELETE, and APPEND. Any update is turned into a RETRIEVE command to isolate tuples to be changed. Revised copies of modified tuples are spooled into a special file. This file is then processed by a "deferred update processor" in process 4, which is discussed in Section 6.

Basically, process 3 performs two functions for RETRIEVE commands. (a) A multivariable query is *decomposed* into a sequence of interactions involving only a single variable. (b) A one-variable query is executed by a one-variable query processor (OVQP). The OVQP in turn performs its function by making calls on the access methods. These two functions are discussed in Section 5; the access methods are indicated in Section 3.

All code to support utility commands (CREATE, DESTROY, INDEX, etc.) resides in process 4. Process 3 simply passes to process 4 any commands which process 4 will execute. Process 4 is organized as a collection of overlays which accomplish the various functions. Some of these functions are discussed in Section 6.

Error messages are passed back through pipes D, E, and F to process 1, which returns them to the user. If the command is a RETRIEVE with no result relation specified, process 3 returns qualifying tuples in a stylized format directly to the "standard output device" of process 1. Unless redirected, this is the user's terminal.

2.3 Invocation from EQUEL

We now turn to the operation of INGRES when invoked by code from the precompiler.

In order to implement EQUEL, a translator (precompiler) was written to convert an EQUEL program into a valid C program with QUEL statements converted to appropriate C code and calls to INGRES. The resulting C program is then compiled by the normal C compiler, producing an executable module. Moreover, when an EQUEL program is run, the executable module produced by the C compiler is used as the front end process in place of the interactive terminal monitor, as noted in Figure 2.

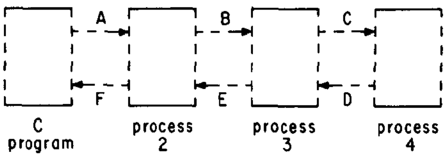

Fig. 2. The forked process structure

During execution of the front end program, database requests (QUEL statements in the EQUEL program) are passed through pipe A and processed by INGRES. Note that unparsed ASCII strings are passed to process 2; the rationale behind this decision is given in [1]. If tuples must be returned for tuple at a time processing, then they are returned through a special data pipe set up between process 3 and the C program. A condition code is also returned through pipe F to indicate success or the type of error encountered.

The functions performed by the EQUEL translator are discussed in detail in [1].

2.4 Comments on the Process Structure

The process structure shown in Figures 1 and 2 is the fourth different process structure implemented. The following considerations suggested this final choice:

(a) Address space limitations. To run on an 11/40, the 64K address space limitation must be adhered to. Processes 2 and 3 are essentially their maximum size; hence they cannot be combined. The code in process 4 is in several overlays because of size constraints.

Were a large address space available, it is likely that processes 2, 3, and 4 would be combined into a single large process. However, the necessity of 3 "core" processes should not degrade performance substantially for the following reasons.

If one large process were resident in main memory, there would be no necessity of swapping code. However, were enough real memory available (~300K bytes) on a UNIX system to hold processes 2 and 3 and all overlays of process 4, no swapping of code would necessarily take place either. Of course, this option is possible only on an 11/70.

On the other hand, suppose one large process was paged into and out of main memory by an operating system and hardware which supported a virtual memory. It is felt that under such conditions page faults would generate I/O activity at approximately the same rate as the swapping/overlaying of processes in INGRES (assuming the same amount of real memory was available in both cases).

Consequently the only sources of overhead that appear to result from multiple processes are the following: (1) Reading or writing pipes require system calls which are considerably more expensive than subroutine calls (which could be used in a single-process system). There are at least eight such system calls needed to execute an INGRES command. (2) Extra code must be executed to format information for transmission on pipes. For example, one cannot pass a pointer to a data structure through a pipe; one must linearize and pass the whole structure.

(b) Simple control flow. The grouping of functions into processes was motivated by the desire for simple control flow. Commands are passed only to the right; data and errors only to the left. Process 3 must issue commands to various overlays in process 4; therefore, it was placed to the left of process 4. Naturally, the parser must precede process 3.

Previous process structures had a more complex interconnection of processes. This made synchronization and debugging much harder.

The structure of process 4 stemmed from a desire to overlay little-used code in a single process. The alternative would have been to create additional processes 5, 6, and 7 (and their associated pipes), which would be quiescent most of the time. This would have required added space in UNIX core tables for no real advantage.

The processes are all synchronized (i.e. each waits for an error return from the next process to the right before continuing to accept input from the process to the left), simplifying the flow of control. Moreover, in many instances the various processes *must* be synchronized. Future versions of INGRES may attempt to exploit parallelism where possible. The performance payoff of such parallelism is unknown at the present time.

(c) Isolation of the front end process. For reasons of protection the C program which replaces the terminal monitor as a front end must run with a user-id different from that of INGRES. Otherwise it could tamper directly with data managed by INGRES. Hence, it must be either overlayed into a process or run in its own process. The latter was chosen for efficiency and convenience.

(d) Rationale for two process structures. The interactive terminal monitor could have been written in EQUEL. Such a strategy would have avoided the existence of two process structures which differ only in the treatment of the data pipe. Since the terminal monitor was written prior to the existence of EQUEL, this option could not be followed. Rewriting the terminal monitor in EQUEL is not considered a high priority task given current resources. Moreover, an EQUEL monitor would be slightly slower because qualifying tuples would be returned to the calling program and then displayed rather than being displayed directly by process 3.

3. DATA STRUCTURES AND ACCESS METHODS

We begin this section with a discussion of the files that INGRES manipulates and their contents. Then we indicate the five possible storage structures (file formats) for relations. Finally we sketch the access methods language used to interface uniformly to the available formats.

3.1 The INGRES File Structure

Figure 3 indicates the subtree of the UNIX file system that INGRES manipulates. The root of this subtree is a directory made for the UNIX user "INGRES." (When

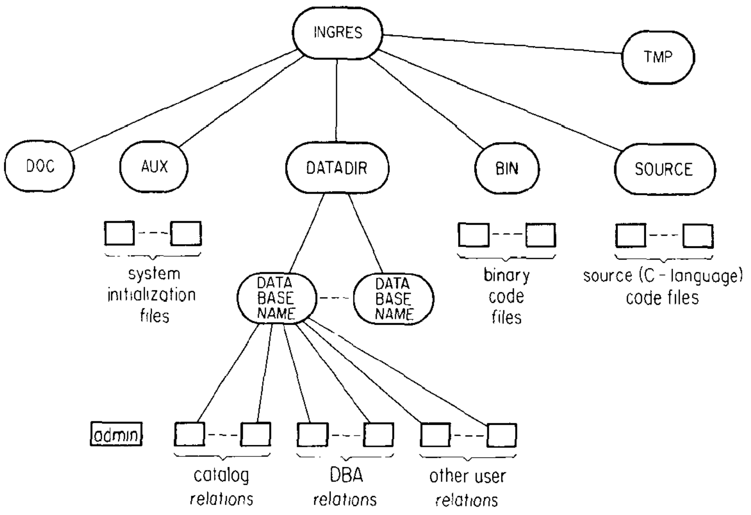

Fig. 3. The INGRES subtree

the INGRES system is initially installed such a user must be created. This user is known as the "superuser" because of the powers available to him. This subject is discussed further in [28].) This root has six descendant directories. The AUX directory has descendant files containing tables which control the spawning of processes (shown in Figures 1 and 2) and an authorization list of users who are allowed to create databases. Only the INGRES superuser may modify these files (by using the UNIX editor). BIN and SOURCE are directories indicating descendant files of respectively object and source code. TMP has descendants which are temporary files for the workspaces used by the interactive terminal monitor. DOC is the root of a subtree with system documentation and the reference manual. Last, there is a directory entry in DATADIR for each database that exists in INGRES. These directories contain the database files in a given database as descendants.

These database files are of four types:

(a) Administration file. This contains the user-id of the database administrator (DBA) and initialization information.

(b) Catalog (system) relations. These relations have predefined names and are created for every database. They are owned by the DBA and constitute the system catalogs. They may be queried by a knowledgeable user issuing RETRIEVE statements; however, they may be updated only by the INGRES utility commands (or directly by the INGRES superuser in an emergency). (When protection statements are implemented the DBA will be able to selectively restrict RETRIEVE access to these relations if he wishes.) The form and content of some of these relations will be discussed presently.

(c) DBA relations. These are relations owned by the DBA and are shared in that any user may access them. When protection is implemented the DBA can "authorize" shared use of these relations by inserting protection predicates (which will be in one of the system relations and may be unique for each user) and deauthorize use by removing such predicates. This mechanism is discussed in [28].

(d) Other relations. These are relations created by other users (by RETRIEVE INTO W or CREATE) and are *not shared.*

Three comments should be made at this time.

(a) The DBA has the following powers not available to ordinary users: the ability to create shared relations and to specify access control for them; the ability to run PURGE; the ability to destroy any relations in his database (except the system catalogs).

This system allows "one-level sharing" in that only the DBA has these powers, and he cannot delegate any of them to others (as in the file systems of most time sharing systems). This strategy was implemented for three reasons: (1) The need for added generality was not perceived. Moreover, added generality would have created tedious problems (such as making revocation of access privileges nontrivial). (2) It seems appropriate to entrust to the DBA the duty (and power) to resolve the policy decision which must be made when space is exhausted and some relations must be destroyed or archived. This policy decision becomes much harder (or impossible) if a database is not in the control of one user. (3) Someone must be entrusted with the policy decision concerning which relations are physically stored and which are defined as "views." This "database design" problem is best centralized in a single DBA.

(b) Except for the single administration file in each database, every file is treated as a relation. Storing system catalogs as relations has the following advantages: (1) Code is economized by sharing routines for accessing both catalog and data relations. (2) Since several storage structures are supported for accessing data relations quickly and flexibly under various interaction mixes, these same storage choices may be utilized to enhance access to catalog information. (3) The ability to execute QUEL statements to examine (and patch) system relations where necessary has greatly aided system debugging.

(c) Each relation is stored in a separate file, i.e. no attempt is made to "cluster" tuples from *different* relations which may be accessed together on the same or on a nearby page.

Note clearly that this clustering is analogous to DBTG systems in declaring a record type to be accessed via a set type which associates records of that record type with a record of a different record type. Current DBTG implementations usually attempt to physically cluster these associated records.

Note also that clustering tuples from one relation in a given file has obvious performance implications. The clustering techniques of this nature that INGRES supports are indicated in Section 3.3.

The decision not to cluster tuples from different relations is based on the following reasoning. (1) UNIX has a small (512-byte) page size. Hence it is expected that the number of tuples which can be grouped on the same page is small. Moreover, logically adjacent pages in a UNIX file are *not necessarily* physically adjacent. Hence clustering tuples on "nearby" pages has no meaning in UNIX; the next logical page in a file may be further away (in terms of disk arm motion) than a page in a different file. In keeping with the design decision of *not* modifying UNIX, these considerations were incorporated in the design decision not to support clustering. (2) The access methods would be more complicated if clustering were supported. (3) Clustering of tuples only makes sense if associated tuples can be linked together using "sets" [6], "links" [29], or some other scheme for identifying clusters. Incorporating these access paths into the decomposition scheme would have greatly increased its complexity.

It should be noted that the designers of System R have reached a different conclusion concerning clustering [2].

3.2 System Catalogs

We now turn to a discussion of the system catalogs. We discuss two relations in detail and indicate briefly the contents of the others.

The RELATION relation contains one tuple for every relation in the database (including all the system relations). The domains of this relation are:

relid	the name of the relation.
owner	the UNIX user-id of the relation owner; when appended to relid it produces a unique file name for storing the relation.
spec	indicates one of five possible storage schemes or else a special code indicating a virtual relation (or "view").
indexd	flag set if secondary index exists for this relation. (This flag and the following two are present to improve performance by avoiding catalog lookups when possible during query modification and one variable query processing.)

protect	flag set if this relation has protection predicates.
integ	flag set if there are integrity constraints.
save	scheduled lifetime of relation.
tuples	number of tuples in relation (kept up to date by the routine "closer" discussed in the next section).
atts	number of domains in relation.
width	width (in bytes) of a tuple.
prim	number of primary file pages for this relation.

The ATTRIBUTE catalog contains information relating to individual domains of relations. Tuples of the ATTRIBUTE catalog contain the following items for each domain of every relation in the database:

relid	name of relation in which attribute appears.
owner	relation owner.
domain_name	domain name.
domain_no	domain number (position) in relation. In processing interactions INGRES uses this number to reference this domain.
offset	offset in bytes from beginning of tuple to beginning of domain.
type	data type of domain (integer, floating point, or character string).
length	length (in bytes) of domain.
keyno	if this domain is part of a key, then "keyno" indicates the ordering of this domain within the key.

These two catalogs together provide information about the structure and content of each relation in the database. No doubt items will continue to be added or deleted as the system undergoes further development. The first planned extensions are the minimum and maximum values assumed by domains. These will be used by a more sophisticated decomposition scheme being developed, which is discussed briefly in Section 5 and in detail in [30]. The representation of the catalogs as relations has allowed this restructuring to occur very easily.

Several other system relations exist which provide auxiliary information about relations. The INDEX catalog contains a tuple for every secondary index in the database. Since secondary indices are themselves relations, they are independently cataloged in the RELATION and ATTRIBUTE relations. However, the INDEX catalog provides the association between a primary relation and its secondary indices and records which domains of the primary relation are in the index.

The PROTECTION and INTEGRITY catalogs contain respectively the protection and integrity predicates for each relation in the database. These predicates are stored in a partially processed form as character strings. (This mechanism exists for INTEGRITY and will be implemented in the same way for PROTECTION.) The VIEW catalog will contain, for each virtual relation, a partially processed QUEL-like description of the view in terms of existing relations. The use of these last three catalogs is described in Section 4. The existence of any of this auxiliary information for a given relation is signaled by the appropriate flag(s) in the RELATION catalog.

Another set of system relations consists of those used by the graphics subsystem to catalog and process maps, which (like everything else) are stored as relations in the database. This topic has been discussed separately in [13].

3.3 Storage Structures Available

We will now describe the five storage structures currently available in INGRES. Four of the schemes are keyed, i.e. the storage location of a tuple within the file is a function of the value of the tuple's key domains. They are termed "hashed," "ISAM," "compressed hash," and "compressed ISAM." For all four structures the key may be any ordered collection of domains. These schemes allow rapid access to specific portions of a relation when key values are supplied. The remaining non-keyed scheme (a "heap") stores tuples in the file independently of their values and provides a low overhead storage structure, especially attractive in situations requiring a complete scan of the relation.

The nonkeyed storage structure in INGRES is a randomly ordered sequential file. Fixed length tuples are simply placed sequentially in the file in the order supplied. New tuples added to the relation are merely appended to the end of the file. The unique tuple identifier for each tuple is its byte-offset within the file. This mode is intended mainly for (a) very small relations, for which the overhead of other schemes is unwarranted; (b) transitional storage of data being moved into or out of the system by COPY; (c) certain temporary relations created as intermediate results during query processing.

In the remaining four schemes the key-value of a tuple determines the page of the file on which the tuple will be placed. The schemes share a common "page-structure" for managing tuples on file pages, as shown in Figure 4.

A tuple must fit entirely on a single page. Its unique tuple identifier (TID) consists of a page number (the ordering of its page in the UNIX file) plus a line number. The line number is an index into a line table, which grows upward from the bottom of the page, and whose entries contain pointers to the tuples on the page. In this way the physical arrangement of tuples on a page can be reorganized without affecting TIDs.

Initially the file contains all its tuples on a number of primary pages. If the relation grows and these pages fill, overflow pages are allocated and chained by pointers

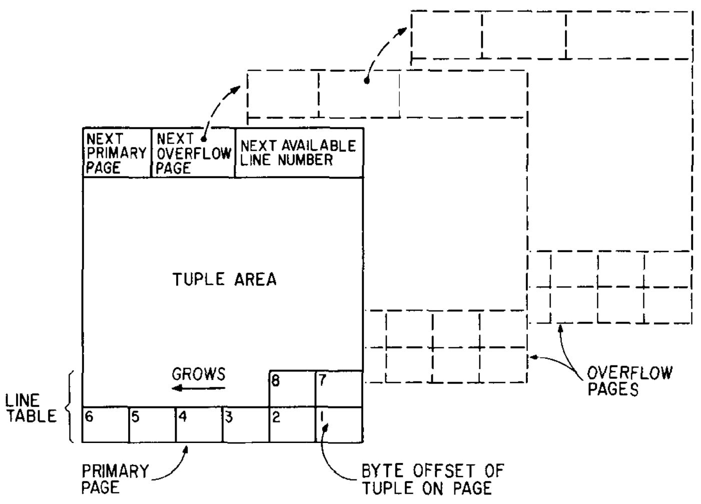

Fig. 4. Page layout for keyed storage structures

to the primary pages with which they are associated. Within a chained group of pages no special ordering of tuples is maintained. Thus in a keyed access which locates a particular primary page, tuples matching the key may actually appear on any page in the chain.

As discussed in [16], two modes of key-to-address transformation are used—randomizing (or "hashing") and order preserving. In a "hash" file tuples are distributed randomly throughout the primary pages of the file according to a hashing function on a key. This mode is well suited for situations in which access is to be conditioned on a specific key value.

As an order preserving mode, a scheme similar to IBM's ISAM [18] is used. The relation is sorted to produce the ordering on a particular key. A multilevel directory is created which records the high key on each primary page. The directory, which is static, resides on several pages following the primary pages within the file itself. A primary page and its overflow pages are *not* maintained in sort order. This decision is discussed in Section 4.2. The "ISAM-like" mode is useful in cases where the key value is likely to be specified as falling within a range of values, since a near ordering of the keys is preserved. The index compression scheme discussed in [16] is currently under implementation.

In the above-mentioned keyed modes, fixed length tuples are stored. In addition, both schemes can be used in conjunction with data compression techniques [14] in cases where increased storage utilization outweighs the added cost of encoding and decoding data during access. These modes are known as "compressed hash" and "compressed ISAM."

The current compression scheme suppresses blanks and portions of a tuple which match the preceding tuple. This compression is applied to each page independently. Other schemes are being experimented with. Compression appears to be useful in storing variable length domains (which must be declared their maximum length). Padding is then removed during compression by the access method. Compression may also be useful when storing secondary indices.

3.4 Access Methods Interface

The Access Methods Interface (AMI) handles all actual accessing of data from relations. The AMI language is implemented as a set of functions whose calling conventions are indicated below. A separate copy of these functions is loaded with each of processes 2, 3, and 4.

Each access method must do two things to support the following calls. First, it must provide some linear ordering of the tuples in a relation so that the concept of "next tuple" is well defined. Second, it must assign to each tuple a unique tuple-id (TID).

The nine implemented calls are as follows:

(a) OPENR(descriptor, mode, relation_name)

Before a relation may be accessed it must be "opened." This function opens the UNIX file for the relation and fills in a "descriptor" with information about the relation from the RELATION and ATTRIBUTE catalogs. The descriptor (storage for which must be declared in the calling routine) is used in subsequent calls on AMI routines as an input parameter to indicate which relation is involved. Consequently, the AMI data accessing routines need not themselves check the system catalogs for the description of a relation. "Mode" specifies whether the relation is being opened for update or for retrieval only.

(b) GET(descriptor, tid, limit_tid, tuple, next_flag)

This function retrieves into "tuple," a single tuple from the relation indicated by "descriptor." "Tid" and "limit_tid" are tuple identifiers. There are two modes of retrieval, "scan" and "direct." In "scan" mode GET is intended to be called successively to retrieve all tuples within a range of tuple-ids. An initial value of "tid" sets the low end of the range desired and "limit_tid" sets the high end. Each time GET is called with "next-flag" = TRUE, the tuple following "tid" is retrieved and its tuple-id is placed into "tid" in readiness for the next call. Reaching "limit_tid" is indicated by a special return code, The initial settings of "tid" and "limit_tid" are done by calling the FIND function. In "direct" mode ("next_flag" = FALSE), GET retrieves the tuple with tuple-id = "tid."

(c) FIND(descriptor, key, tid, key_type)

When called with a negative "key-type," FIND returns in "tid" the lowest tuple-id on the lowest page which could possibly contain tuples matching the key supplied. Analogously, the highest tuple-id is returned when "key-type" is positive. The objective is to restrict the scan of a relation by eliminating tuples from consideration which are known from their placement not to satisfy a given qualification.

"Key-type" also indicates (through its absolute value) whether the key, if supplied, is an EXACTKEY or a RANGEKEY. Different criteria for matching are applied in each case. An EXACTKEY matches only those tuples containing exactly the value of the key supplied. A RANGEKEY represents the low (or high) end of a range of possible key values and thus matches any tuple with a key value greater than or equal to (or less than or equal to) the key supplied. Note that only with an order preserving storage structure can a RANGEKEY be used to successfully restrict a scan.

In cases where the storage structure of the relation is incompatible with the "key-type," the "tid" returned will be as if no key were supplied (that is, the lowest or highest tuple in the relation). Calls to FIND invariably occur in pairs, to obtain the two tuple-ids which establish the low and high ends of the scan done in subsequent calls to GET.

Two functions are available for determining the access characteristics of the storage structure of a primary data relation or secondary index, respectively.

(d) PARAMD(descriptor, access_characteristics_structure)

(e) PARAMI(index-descriptor, access_characteristics_structure)

The "access-characteristics-structure" is filled in with information regarding the type of key which may be utilized to restrict the scan of a given relation: It indicates whether exact key values or ranges of key values can be used, and whether a partially specified key may be used. This determines the "key-type" used in a subsequent call to FIND. The ordering of domains in the key is also indicated. These two functions allow the access optimization routines to be coded independently of the specific storage structures currently implemented.

Other AMI functions provide a facility for updating relations.

(f) INSERT(descriptor, tuple)

The tuple is added to the relation in its "proper" place according to its key value and the storage mode of the relation.

(g) REPLACE(descriptor, tid, new_tuple)

(h) DELETE(descriptor, tid)

The tuple indicated by "tid" is either replaced by new values or deleted from the relation altogether. The tuple-id of the affected tuple will have been obtained by a previous GET.

Finally, when all access to a relation is complete it must be closed:

(i) CLOSER(descriptor)

This closes the relation's UNIX file and rewrites the information in the descriptor back into the system catalogs if there has been any change.

3.5 Addition of New Access Methods

One of the goals of the AMI design was to insulate higher level software from the actual functioning of the access methods, thereby making it easier to add different ones. It is anticipated that users with special requirements will take advantage of this feature.

In order to add a new access method, one need only extend the AMI routines to handle the new case. If the new method uses the same page layout and TID scheme, only FIND, PARAMI, and PARAMD need to be extended. Otherwise new procedures to perform the mapping of TIDs to physical file locations must be supplied for use by GET, INSERT, REPLACE, and DELETE.

4. THE STRUCTURE OF PROCESS 2

Process 2 contains four main components:

(a) a lexical analyzer;
(b) a parser (written in YACC [19]);
(c) concurrency control routines;
(d) query modification routines to support protection, views, and integrity control (at present only partially implemented).

Since (a) and (b) are designed and implemented along fairly standard lines, only (c) and (d) will be discussed in detail. The output of the parsing process is a tree structured representation of the input query used as the internal form in subsequent processing. Furthermore, the qualification portion of the query has been converted to an equivalent Boolean expression in conjunctive normal form. In this form the query tree is then ready to undergo what has been termed "query modification."

4.1 Query Modification

Query modification includes adding integrity and protection predicates to the original query and changing references to virtual relations into references to the appropriate physical relations. At the present time only a simple integrity scheme has been implemented.

In [27] algorithms of several levels of complexity are presented for performing integrity control on updates. In the present system only the simplest case, involving single-variable, aggregate free integrity assertions, has been implemented, as described in detail in [23].

Briefly, integrity assertions are entered in the form of QUEL qualification clauses to be applied to interactions updating the relation over which the variable in the assertion ranges. A parse tree is created for the qualification and a representation of this tree is stored in the INTEGRITY catalog together with an indication of the relation and the specific domains involved. At query modification time, updates are checked for any possible integrity assertions on the affected domains. Relevant assertions are retrieved, rebuilt into tree form, and grafted onto the update tree so as to AND the assertions with the existing qualification of the interaction.

Algorithms for the support of views are also given in [27]. Basically a view is a virtual relation defined in terms of relations which physically exist. Only the view definition will be stored, and it will be indicated to INGRES by a DEFINE command. This command will have a syntax identical to that of a RETRIEVE statement. Thus legal views will be those relations which it is possible to materialize by a RETRIEVE statement. They will be allowed in INGRES to support EQUEL programs written for obsolete versions of the database and for user convenience.

Protection will be handled according to the algorithm described in [25]. Like integrity control, this algorithm involves adding qualifications to the user's interaction. The details of the implementation (which is in progress) are given in [28], which also includes a discussion of the mechanisms being implemented to physically protect INGRES files from tampering in any way other than by executing the INGRES object code. Last, [28] distinguishes the INGRES protection scheme from the one based on views in [5] and indicates the rationale behind its use.

In the remainder of this section we give an example of query modification at work.

Suppose at a previous point in time all employees in the EMPLOYEE relation were under 30 and had no manager recorded. If an EQUEL program had been written for this previous version of EMPLOYEE which retrieved ages of employees coded into 5 bits, it would now fail for employees over 31.

If one wishes to use the above program without modification, then the following view must be used:

```
RANGE OF E IS EMPLOYEE
DEFINE OLDEMP (E.NAME, E.DEPT, E.SALARY, E.AGE)
WHERE  E.AGE < 30
```

Suppose that all employees in the EMPLOYEE relation must make more than $8000. This can be expressed by the integrity constraint:

```
RANGE OF E IS EMPLOYEE
INTEGRITY CONSTRAINT IS E.SALARY > 8000
```

Last, suppose each person is only authorized to alter salaries of employees whom he manages. This is expressed as follows:

```
RANGE OF E IS EMPLOYEE
PROTECT EMPLOYEE FOR ALL (E.SALARY; E.NAME)
WHERE  E.MANAGER = *
```

The * is a surrogate for the logon name of the current UNIX user of INGRES. The semicolon separates updatable from nonupdatable (but visible) domains.

Suppose Smith through an EQUEL program or from the terminal monitor issues the following interaction:

```
RANGE OF L IS OLDEMP
REPLACE L(SALARY = .9*L.SALARY)
WHERE  L.NAME = "Brown"
```

This is an update on a view. Hence the view algorithm in [27] will first be applied to yield:

```
RANGE OF E IS EMPLOYEE
REPLACE E(SALARY = .9*E.SALARY)
WHERE  E.NAME = "Brown"
AND  E.AGE < 30
```

Note Brown is only in OLDEMP if he is under 30. Now the integrity algorithm in [27] must be applied to ensure that Brown's salary is not being cut to as little as $8000. This involves modifying the interaction to:

```
RANGE OF E IS EMPLOYEE
REPLACE E(SALARY = .9*E.SALARY)
WHERE  E.NAME = "Brown"
   AND  E.AGE < 30
   AND  .9*E.SALARY > $8000
```

Since .9*E.SALARY will be Brown's salary after the update, the added qualification ensures this will be more than $8000.

Last, the protection algorithm of [28] is applied to yield:

```
RANGE OF E IS EMPLOYEE
REPLACE E(SALARY = .9*E.SALARY)
WHERE  E.NAME = "Brown"
   AND  E.AGE < 30
   AND  .9*E.SALARY > $8000
   AND  E.MANAGER = "Smith"
```

Notice that in all three cases more qualification is ANDed onto the user's interaction. The view algorithm must in addition change tuple variables.

In all cases the qualification is obtained from (or is an easy modification of) predicates stored in the VIEW, INTEGRITY, and PROTECTION relations. The tree representation of the interaction is simply modified to AND these qualifications (which are all stored in parsed form).

It should be clearly noted that only one-variable, aggregate free integrity assertions are currently supported. Moreover, even this feature is not in the released version of INGRES. The code for both concurrency control and integrity control will not fit into process 2 without exceeding 64K words. The decision was made to release a system with concurrency control.

The INGRES designers are currently adding a fifth process (process 2.5) to hold concurrency and query modification routines. On PDP 11/45s and 11/70s that have a 128K address space this extra process will not be required.

4.2 Concurrency Control

In any multiuser system provisions must be included to ensure that multiple concurrent updates are executed in a manner such that some level of data integrity can be guaranteed. The following two updates illustrate the problem.

```
     RANGE OF E IS EMPLOYEE
U1   REPLACE E(DEPT = "toy")
       WHERE E.DEPT = "candy"

     RANGE OF F IS EMPLOYEE
U2   REPLACE F(DEPT = "candy")
       WHERE F.DEPT = "toy"
```

If U1 and U2 are executed concurrently with no controls, some employees may end up in each department and the particular result may not be repeatable if the database is backed up and the interactions reexecuted.

The control which must be provided is to guarantee that some database operation is "atomic" (occurs in such a fashion that it *appears* instantaneous and before or after any other database operation). This atomic unit will be called a "transaction."

In INGRES there are five basic choices available for defining a transaction:

(a) something smaller than one INGRES command;
(b) one INGRES command;
(c) a collection of INGRES commands with no intervening C code;
(d) a collection of INGRES commands with C code but no system calls;
(e) an arbitrary EQUEL program.

If option (a) is chosen, INGRES could not guarantee that two concurrently executing update commands would give the same result as if they were executed sequentially (in either order) in one collection of INGRES processes. In fact, the outcome could fail to be repeatable, as noted in the example above. This situation is clearly undesirable.

Option (e) is, in the opinion of the INGRES designers, impossible to support. The following transaction could be declared in an EQUEL program.

```
BEGIN TRANSACTION
    FIRST QUEL UPDATE
    SYSTEM CALLS TO CREATE AND DESTROY FILES
    SYSTEM CALLS TO FORK A SECOND COLLECTION OF INGRES PROCESSES
        TO WHICH COMMANDS ARE PASSED
    SYSTEM CALLS TO READ FROM A TERMINAL
    SYSTEM CALLS TO READ FROM A TAPE
    SECOND QUEL UPDATE (whose form depends on previous two system calls)
END TRANSACTION
```

Suppose T1 is the above transaction and runs concurrently with a transaction T2 involving commands of the same form. The second update of each transaction may well conflict with the first update of the other. Note that there is no way to tell a priori that T1 and T2 conflict, since the form of the second update is not known in advance. Hence a deadlock situation can arise which can only be resolved by aborting one transaction (an undesirable policy in the eyes of the INGRES designers) or attempting to back out one transaction. The overhead of backing out through the intermediate system calls appears prohibitive (if it is possible at all).

Restricting a transaction to have no system calls (and hence no I/O) cripples the power of a transaction in order to make deadlock resolution possible. This was judged undesirable.

For example, the following transaction requires such system calls:

```
BEGIN TRANSACTION
    QUEL RETRIEVE to find all flights on a particular day from San Francisco to Los
        Angeles with space available.
    Display flights and times to user.
    Wait for user to indicate desired flight.
    QUEL REPLACE to reserve a seat on the flight of the user's choice.
END TRANSACTION
```

If the above set of commands is not a transaction, then space on a flight may not be available when the REPLACE is executed even though it was when the RETRIEVE occurred.

Since it appears impossible to support multi-QUEL statement transactions (except in a crippled form), the INGRES designers have chosen Option (b), one QUEL statement, as a transaction.

Option (c) can be handled by a straightforward extension of the algorithms to follow and will be implemented if there is sufficient user demand for it. This option can support "triggers" [2] and may prove useful.

Supporting Option (d) would considerably increase system complexity for what is perceived to be a small generalization. Moreover, it would be difficult to enforce in the EQUEL translator unless the translator parsed the entire C language.

The implementation of (b) or (c) can be achieved by physical locks on data items, pages, tuples, domains, relations, etc. [12] or by predicate locks [26]. The current implementation is by relatively crude physical locks (on domains of a relation) and avoids deadlock by not allowing an interaction to proceed to process 3 until it can lock all required resources. Because of a problem with the current design of the REPLACE access method call, all domains of a relation must currently be locked (i.e. a whole relation is locked) to perform an update. This situation will soon be rectified.

The choice of avoiding deadlock rather than detecting and resolving it is made primarily for implementation simplicity.

The choice of a crude locking unit reflects our environment where core storage for a large lock table is not available. Our current implementation uses a LOCK relation into which a tuple for each lock requested is inserted. This entire relation is physically locked and then interrogated for conflicting locks. If none exist, all needed locks are inserted. If a conflict exists, the concurrency processor "sleeps" for a fixed interval and then tries again. The necessity to lock the entire relation and to sleep for a fixed interval results from the absence of semaphores (or an equivalent mechanism) in UNIX. Because concurrency control can have high overhead as currently implemented, it can be turned off.

The INGRES designers are considering writing a device driver (a clean extension to UNIX routinely written for new devices) to alleviate the lack of semaphores. This driver would simply maintain core tables to implement desired synchronization and physical locking in UNIX.

The locks are held by the concurrency processor until a termination message is received on pipe E. Only then does it delete its locks.

In the future we plan to experimentally implement a crude (and thereby low CPU overhead) version of the predicate locking scheme described in [26]. Such an approach may provide considerable concurrency at an acceptable overhead in lock table space and CPU time, although such a statement is highly speculative.

To conclude this section, we briefly indicate the reasoning behind not sorting a page and its overflow pages in the "ISAM-like" access method. This topic is also discussed in [17].

The proposed device driver for locking in UNIX must at least ensure that read-modify-write of a single UNIX page is an atomic operation. Otherwise, INGRES would still be required to lock the whole LOCK relation to insert locks. Moreover, any proposed predicate locking scheme could not function without such an atomic operation. If the lock unit is a UNIX page, then INGRES can insert and delete a tuple from a relation by holding only one lock at a time if a primary page and its overflow page are unordered. However, maintenance of the sort order of these pages may require the access method to lock more than one page when it inserts a tuple. Clearly deadlock may be possible given concurrent updates, and the size of the lock table in the device driver is not predictable. To avoid both problems these pages remain unsorted.

5. PROCESS 3

As noted in Section 2, this process performs the following two functions, which will be discussed in turn:

(a) Decomposition of queries involving more than one variable into sequences of one-variable queries. Partial results are accumulated until the entire query is evaluated. This program is called DECOMP. It also turns any updates into the appropriate queries to isolate qualifying tuples and spools modifications into a special file for deferred update.

(b) Processing of single-variable queries. The program is called the one-variable query processor (OVQP).

5.1 DECOMP

Because INGRES allows interactions which are defined on the crossproduct of perhaps several relations, efficient execution of this step is of crucial importance in searching as small a portion of the appropriate crossproduct space as possible. DECOMP uses three techniques in processing interactions. We describe each technique, and then give the actual algorithm implemented followed by an example which illustrates all features. Finally we indicate the role of a more sophisticated decomposition scheme under design.

(a) Tuple substitution. The basic technique used by DECOMP to reduce a query to fewer variables is tuple substitution. One variable (out of possibly many) in the query is selected for substitution. The AMI language is used to scan the relation associated with the variable one tuple at a time. For each tuple the values of domains in that relation are substituted into the query. In the resulting modified query, all previous references to the substituted variable have now been replaced by values (constants) and the query has thus been reduced to one less variable. Decomposition is repeated (recursively) on the modified query until only one variable remains, at which point the OVQP is called to continue processing.

(b) One-variable detachment. If the qualification Q of the query is of the form

$Q_1(V_1)$ AND $Q_2(V_1, \ldots, V_n)$

for some tuple variable V_1, the following two steps can be executed:

(1) Issue the query
RETRIEVE INTO W (TL[V_1])
WHERE $Q_1[V_1]$

Here TL[V_1] are those domains required in the remainder of the query. Note that this is a one-variable query and may be passed directly to OVQP.

(2) Replace R_1, the relation over which V_1 ranges, by W in the range declaration and delete $Q_1[V_1]$ from Q.

The query formed in step 1 is called a "one-variable, detachable subquery," and the technique for forming and executing it is called "one-variable detachment" (OVD). This step has the effect of reducing the size of the relation over which V_1 ranges by restriction and projection. Hence it may reduce the complexity of the processing to follow.

Moreover, the opportunity exists in the process of creating new relations through OVD, to choose storage structures, and particularly keys, which will prove helpful in further processing.

(c) Reformatting. When a tuple variable is selected for substitution, a large number of queries, each with one less variable, will be executed. If (b) is a possible operation after the substitution for some remaining variable V_1, then the relation over which V_1 ranges, R_1, can be reformatted to have domains used in $Q_1(V_1)$ as a key. This will expedite (b) each time it is executed during tuple substitution.

We can now state the complete decomposition algorithm. After doing so, we illustrate all steps with an example.

Step 1. If the number of variables in the query is 0 or 1, call OVQP and then return; else go on to step 2.

Step 2. Find all variables, $\{V_1, \ldots, V_n\}$, for which the query contains a one-variable clause. Perform OVD to create new ranges for each of these variables. The new relation for each variable V_i is stored as a hash file with key K_i chosen as follows:

2.1. For each j select from the remaining multivariable clauses in the query the collection, C_{ij}, which have the form $V_i \cdot d_i = V_j \cdot d_j$, where d_i, d_j are domains of V_i and V_j.

2.2. From the key K_i to be the concatenation of domains d_{i1}, d_{i2}, . . . of V_i appearing in clauses in C_{ij}.

2.3. If more than one j exists, for which C_{ij} is nonempty, one C_{ij} is chosen arbitrarily for forming the key. If C_{ij} is empty for all j, the relation is stored as an unsorted table.

Step 3. Choose the variable V_s with the smallest number of tuples as the next one for which to perform tuple substitution.

Step 4. For each tuple variable V_j for which C_{js} is nonnull, reformat if necessary the storage structure of the relation R_j over which it ranges so that the key of R_j is the concatenation of domains d_{j1}, . . . appearing in C_{js}. This ensures that when the clauses in C_{js} become one-variable after substituting for V_s, subsequent calls to OVQP to restrict further the range of V_j will be done as efficiently as possible.

Step 5. Iterate the following steps over all tuples in the range of the variable selected in step 3 and then return:

5.1. Substitute values from tuple into query.

5.2. Invoke decomposition algorithm recursively on a copy of resulting query which now has been reduced by one variable.

5.3. Merge the results from 5.2 with those of previous iterations.

We use the following query to illustrate the algorithm:

```
RANGE OF E, M IS EMPLOYEE
RANGE OF D IS DEPT
RETRIEVE (E.NAME)
WHERE  E.SALARY   > M.SALARY  AND
       E.MANAGER  = M.NAME    AND
       E.DEPT     = D.DEPT    AND
       D.FLOOR#   = 1         AND
       E.AGE      > 40
```

This request is for employees over 40 on the first floor who earn more than their manager.

LEVEL 1

Step 1. Query is not one variable.

Step 2. Issue the two queries:

```
RANGE OF D IS DEPT
RETRIEVE INTO T1(D.DEPT)                                          (1)
WHERE  D.FLOOR# = 1

RANGE OF E IS EMPLOYEE
RETRIEVE INTO T2(E.NAME, E.SALARY, E.MANAGER, E.DEPT)             (2)
WHERE E.AGE > 40
```

T1 is stored hashed on DEPT; however, the algorithm must choose arbitrarily between hashing T2 on MANAGER or DEPT. Suppose it chooses MANAGER. The original query now becomes:

```
RANGE OF D IS T1
RANGE OF E IS T2
RANGE OF M IS EMPLOYEE
RETRIEVE (E.NAME)
WHERE  E.SALARY   > M.SALARY  AND
       E.MANAGER  = M.NAME    AND
       E.DEPT     = D.DEPT
```

Step 3. Suppose T1 has smallest cardinality. Hence D is chosen for substitution.

Step 4. Reformat T2 to be hashed on DEPT; the guess chosen in step 2 above was a poor one.

Step 5. Iterate for each tuple in T1 and then quit:

5.1 Substitute value for D. DEPT yielding

```
RANGE OF E IS T1
RANGE OF M IS EMPLOYEE
RETRIEVE (E.NAME)
WHERE  E.SALARY > M.SALARY    AND
       E.MANAGER = M.NAME     AND
       E.DEPT     = value
```

5.2. Start at step 1 with the above query as input (Level 2 below).

5.3. Cumulatively merge results as they are obtained.

LEVEL 2

Step 1. Query is not one variable.

Step 2. Issue the query

```
    RANGE OF E IS T2
    RETRIEVE INTO T3 (E.NAME, E.SALARY, E.NAME)                (3)
    WHERE  E.DEPT = value
```

T3 is constructed hashed on MANAGER. T2 in step 4 in Level 1 above is reformatted so that this query (which will be issued once for each tuple in T1) will be done efficiently by OVQP. Hopefully the cost of reformatting is small compared to the savings at this step. What remains is

```
    RANGE OF E IS T3
    RANGE IF M IS EMPLOYEE
    RETRIEVE (E.NAME)
    WHERE  E.SALARY    > M.SALARY  AND
           E.MANAGER = M.NAME
```

Step 3. T3 has less tuples than EMPLOYEE; therefore choose T3.

Step 4. [unnecessary]

Step 5. Iterate for each tuple in T3 and then return to previous level:
  5.1. Substitute values for E.NAME, E.SALARY, and E.MANAGER, yielding

```
    RANGE OF M IS EMPLOYEE
    RETRIEVE (VALUE 1)                                         (4)
    WHERE  Value2 > M.SALARY  AND
           Value3 = M.NAME
```

  5.2. Start at step 1 with this query as input (Level 3 below).
  5.3. Cumulatively merge results as obtained.

LEVEL 3

Step 1. Query has one variable; invoke OVQP and then return to previous level.

The algorithm thus decomposes the original query into the four prototype, one-variable queries labeled (1)–(4), some of which are executed repetitively with different constant values and with results merged appropriately. Queries (1) and (2) are executed once, query (3) once for each tuple in T1, and query (4) the number of times equal to the number of tuples in T1 times the number of tuples in T3.

The following comments on the algorithm are appropriate.

(a) OVD is almost always assured of speeding processing. Not only is it possible to choose the storage structure of a temporary relation wisely, but also the cardinality of this relation may be much less than the one it replaces as the range for a tuple variable. It only fails if little or no reduction takes place and reformatting is unproductive.

It should be noted that a temporary relation is created rather than a list of qualifying tuple-id's. The basic tradeoff is that OVD must copy qualifying tuples but can remove duplicates created during the projection. Storing tuple-id's avoids the copy operation at the expense of reaccessing qualifying tuples and retaining duplicates. It is clear that cases exist where each strategy is superior. The INGRES designers have chosen OVD because it does not appear to offer worse performance than the alternative, allows a more accurate choice of the variable with the smallest range in step 3 of the algorithm above, and results in cleaner code.

(b) Tuple substitution is done when necessary on the variable associated with the smallest number of tuples. This has the effect of reducing the number of eventual calls on OVQP.

(c) Reformatting is done (if necessary) with the knowledge that it will usually replace a collection of complete sequential scans of a relation by a collection of limited scans. This almost always reduces processing time.

(d) It is believed that this algorithm efficiently handles a large class of interactions. Moreover, the algorithm does not require excessive CPU overhead to perform. There are, however, cases where a more elaborate algorithm is indicated. The following comment applies to such cases.

(e) Suppose that we have two or more strategies ST_0, ST_1, . . . , ST_n, each one being better than the previous one but also requiring a greater overhead. Suppose further that we begin an interaction on ST_0 and run it for an amount of time equal to a fraction of the estimated overhead of ST_1. At the end of that time, by simply counting the number of tuples of the first substitution variable which have already been processed, we can get an estimate for the total processing time using ST_0. If this is significantly greater than the overhead of ST_1, then we switch to ST_1. Otherwise we stay and complete processing the interaction using ST_0. Obviously, the procedure can be repeated on ST_1 to call ST_2 if necessary, and so forth.

The algorithm detailed in this section may be thought of as ST_0. A more sophisticated algorithm is currently under development [30].

5.2 One-Variable Query Processor (OVQP)

This module is concerned solely with the efficient accessing of tuples from a single relation given a particular one-variable query. The initial portion of this program, known as STRATEGY, determines what key (if any) may be used profitably to access the relation, what value(s) of that key will be used in calls to the AMI routine FIND, and whether access may be accomplished directly through the AMI to the storage structure of the primary relation itself or if a secondary index on the relation should be used. If access is to be through a secondary index, then STRATEGY must choose which *one* of possibly many indices to use.

Tuples are then retrieved according to the access strategy selected and are processed by the SCAN portion of OVQP. These routines evaluate each tuple against the qualification part of the query, create target list values for qualifying tuples, and dispose of the target list appropriately.

Since SCAN is relatively straightforward, we discuss only the policy decisions made in STRATEGY.

First STRATEGY examines the qualification for clauses which specify the value of a domain, i.e. clauses of the form

V.domain op constant

or

constant op *V*.domain

where "op" is one of the set $\{=, <, >, \leq, \geq\}$. Such clauses are termed "simple" clauses and are organized into a list. The constants in simple clauses will determine the key values input to FIND to limit the ensuing scan.

Obviously a nonsimple clause may be equivalent to a simple one. For example, E.SALARY/2 = 10000 is equivalent to E.SALARY = 20000. However, recognizing and converting such clauses requires a general algebraic symbol manipulator. This issue has been avoided by ignoring all nonsimple clauses.

STRATEGY must select one of two accessing strategies: (a) issuing two AMI FIND commands on the primary relation followed by a sequential scan of the relation (using GET in "scan" mode) between the limits set, or (b) issuing two AMI FIND commands on some index relation followed by a sequential scan of the index between the limits set. For each tuple retrieved the "pointer" domain is obtained; this is simply the tuple-id of a tuple in the primary relation. This tuple is fetched (using GET in "direct" mode) and processed.

To make the choice, the access possibilities available must be determined. Keying information about the primary relation is obtained using the AMI function PARAMD. Names of indices are obtained from the INDEX catalog and keying information about indices is obtained with the function PARAMI.

Further, a compatability between the available access possibilities and the specification of key values by simple clauses must be established. A hashed relation requires that a simple clause specify equality as the operator in order to be useful; for combined (multidomain) keys, all domains must be specified. ISAM structures, on the other hand, allow range specifications; additionally, a combined ISAM key requires only that the most significant domains be specified.

STRATEGY checks for such a compatability according to the following priority order of access possibilities: (1) hashed primary relation, (2) hashed index, (3) ISAM primary relation, (4) ISAM index. The rationale for this ordering is related to the expected number of page accesses required to retrieve a tuple from the source relation in each case. In the following analysis the effect of overflow pages is ignored (on the assumption that the four access possibilities would be equally affected).

In case (1) the key value provided locates a desired source tuple in one access via calculation involving a hashing function. In case (2) the key value similarly locates an appropriate index relation tuple in one access, but an additional access is required to retrieve the proper primary relation tuple. For an ISAM-structured scheme a directory must be examined. This lookup itself incurs at least one access but possibly more if the directory is multilevel. Then the tuple itself must be accessed. Thus case (3) requires at least two (but possibly more) total accesses. In case (4) the use of an index necessitates yet another access in the primary relation, making the total at least three.

To illustrate STRATEGY, we indicate what happens to queries (1)–(4) from Section 5.1.

Suppose EMPLOYEE is an ISAM relation with a key of NAME, while DEPT is hashed on FLOOR#. Moreover a secondary index for AGE exists which is hashed on AGE, and one for SALARY exists which uses ISAM with a key of SALARY.

Query (1): One simple clause exists (D.FLOOR# = 2). Hence Strategy (a) is applied against the hashed primary relation.

Query (2): One simple clause exists (E.AGE > 40). However, it is not usable to limit the scan on a hashed index. Hence a complete (unkeyed) scan of EMPLOYEE is required. Were the index for AGE an ISAM relation, then Strategy (b) would be used on this index.

Query (3): One simple clause exists and T1 has been reformatted to allow Strategy (a) against the hashed primary relation.

Query (4): Two simple clauses exist (value2 > M.SALARY; value3 = M.NAME). Strategy (a) is available on the hashed primary relation, as is Strategy (b) for the ISAM index. The algorithm chooses Strategy (a).

6. UTILITIES IN PROCESS 4

6.1 Implementation of Utility Commands

We have indicated in Section 1 several database utilities available to users. These commands are organized into several overlay programs as noted previously. Bringing the required overlay into core as needed is done in a straightforward way.

Most of the utilities update or read the system relations using AMI calls. MODIFY contains a sort routine which puts tuples in collating sequence according to the concatenation of the desired keys (which need not be of the same data type). Pages are initially loaded to approximately 80 percent of capacity. The sort routine is a recursive *N*-way merge-sort where *N* is the maximum number of files process 4 can have open at once (currently eight). The index building occurs in an obvious way. To convert to hash structures, MODIFY must specify the number of primary pages to be allocated. This parameter is used by the AMI in its hash scheme (which is a standard modulo division method).

It should be noted that a user who creates an empty hash relation using the CREATE command and then copies a large UNIX file into it using COPY creates a very inefficient structure. This is because a relatively small default number of primary pages will have been specified by CREATE, and overflow chains will be long. A better strategy is to COPY into an unsorted table so that MODIFY can subsequently make a good guess at the number of primary pages to allocate.

6.2 Deferred Update and Recovery

Any updates (APPEND, DELETE, REPLACE) are processed by writing the tuples to be added, changed, or modified into a temporary file. When process 3 finishes, it calls process 4 to actually perform the modifications requested and any updates to secondary indices which may be required as a final step in processing. Deferred update is done for four reasons.

(a) Secondary index considerations. Suppose the following QUEL statement is executed:

```
RANGE OF E IS EMPLOYEE
REPLACE E(SALARY = 1.1*E.SALARY)
WHERE  E.SALARY > 20000
```

Suppose further that there is a secondary index on the salary domain and the primary relation is keyed on another domain.

OVQP, in finding the employees who qualify for the raise, will use the secondary index. If one employee qualifies and his tuple is modified and the secondary index updated, then the scan of the secondary index will find his tuple a second time since it has been moved forward. (In fact, his tuple will be found an arbitrary number of times.) Either secondary indexes cannot be used to identify qualifying tuples when range qualifications are present (a rather unnatural restriction), or secondary indices must be updated in deferred mode.

(b) Primary relation considerations. Suppose the QUEL statement

```
RANGE OF E, M IS EMPLOYEE
REPLACE E(SALARY = .9*E.SALARY)
Where  E.MGR     = M.NAME  AND
       E.SALARY > M.SALARY
```

is executed for the following EMPLOYEE relation:

NAME	SALARY	MANAGER
Smith	10K	Jones
Jones	8K	
Brown	9.5K	Smith

Logically Smith should get the pay cut and Brown should not. However, if Smith's tuple is updated before Brown is checked for the pay cut, Brown will qualify. This undesirable situation must be avoided by deferred update.

(c) Functionality of updates. Suppose the following QUEL statement is executed:

```
RANGE OF E, M IS EMPLOYEE
REPLACE E(SALARY = M.SALARY)
```

This update attempts to assign to each employee the salary of every other employee, i.e. a single data item is to be replaced by multiple values. Stated differently, the REPLACE statement does not specify a function. In certain cases (such as a REPLACE involving only one tuple variable) functionality is guaranteed. However, in general the functionality of an update is data dependent. This nonfunctionality can only be checked if deferred update is performed.

To do so, the deferred update processor must check for duplicate TIDs in REPLACE calls (which requires sorting or hashing the update file). This potentially expensive operation does not exist in the current implementation, but will be optionally available in the future.

(d) Recovery considerations. The deferred update file provides a log of updates to be made. Recovery is provided upon system crash by the RESTORE command. In this case the deferred update routine is requested to destroy the temporary file if it has not yet started processing it. If it has begun processing, it reprocesses the entire update file in such a way that the effect is the same as if it were processed exactly once from start to finish.

Hence the update is "backed out" if deferred updating has not yet begun; otherwise it is processed to conclusion. The software is designed so the update file can be optionally spooled onto tape and recovered from tape. This added feature should soon be operational.

If a user from the terminal monitor (or a C program) wishes to stop a command he can issue a "break" character. In this case all processes reset except the deferred update program, which recovers in the same manner as above.

All update commands do deferred update; however the INGRES utilities have not yet been modified to do likewise. When this has been done, INGRES will recover from all crashes which leave the disk intact. In the meantime there can be disk-intact crashes which cannot be recovered in this manner (if they happen in such a way that the system catalogs are left inconsistent).

The INGRES "superuser" can checkpoint a database onto tape using the UNIX backup scheme. Since INGRES logs all interactions, a consistent system can always be obtained, albeit slowly, by restoring the last checkpoint and running the log of interactions (or the tape of deferred updates if it exists).

It should be noted that deferred update is a very expensive operation. One INGRES user has elected to have updates performed directly in process 3, cognizant that he must avoid executing interactions which will run incorrectly. Like checks for functionality, direct update may be optionally available in the future. Of course, a different recovery scheme must be implemented.

7. CONCLUSION AND FUTURE EXTENSIONS

The system described herein is in use at about fifteen installations. It forms the basis of an accounting system, a system for managing student records, a geodata system, a system for managing cable trouble reports and maintenance calls for a large telephone company, and assorted other smaller applications. These applications have been running for periods of up to nine months.

7.1 Performance

At this time no detailed performance measurements have been made, as the current version (labeled Version 5) has been operational for less than two months. We have instrumented the code and are in the process of collecting such measurements.

The sizes (in bytes) of the processes in INGRES are indicated below. Since the access methods are loaded with processes 2 and 3 and with many of the utilities, their contribution to the respective process sizes has been noted separately.

access methods (AM)	11K
terminal monitor	10K
EQUEL	30K + AM
process 2	45K + AM
process 3 (query processor)	45K + AM
utilities (8 overlays)	160K + AM

7.2 User Feedback

The feedback from internal and external users has been overwhelmingly positive. In this section we indicate features that have been suggested for future systems.

(a) Improved performance. Earlier versions of INGRES were very slow; the current version should alleviate this problem.

(b) Recursion. QUEL does not support recursion, which must be tediously programmed in C using the precompiler; recursion capability has been suggested as a desired extension.

(c) Other language extensions. These include user defined functions (especially counters), multiple target lists for a single qualification statement, and if-then-else control structures in QUEL; these features may presently be programmed, but only very inefficiently, using the precompiler.

(d) Report generator. PRINT is a very primitive report generator and the need for augmented facilities in this area is clear; it should be written in EQUEL.

(e) Bulk copy. The COPY routine fails to handle easily all situations that arise.

7.3 Future Extensions

Noted throughout the paper are areas where system improvement is in progress, planned, or desired by users. Other areas of extension include: (a) a multicomputer system version of INGRES to operate on distributed databases; (b) further performance enhancements; (c) a higher level user language including recursion and user defined functions; (d) better data definition and integrity features; and (e) a database administrator advisor.

The database administrator advisor program would run at idle priority and issue queries against a statistics relation to be kept by INGRES. It could then offer advice to a DBA concerning the choice of access methods and the selection of indices. This topic is discussed further in [16].

ACKNOWLEDGMENT

The following persons have played active roles in the design and implementation of INGRES: Eric Allman, Rick Berman, Jim Ford, Angela Go, Nancy McDonald, Peter Rubinstein, Iris Schoenberg, Nick Whyte, Carol Williams, Karel Youssefi, and Bill Zook.

REFERENCES

1. ALLMAN, E., STONEBRAKER, M., AND HELD, G. Embedding a relational data sublanguage in a general purpose programming language. Proc. Conf. on Data, SIGPLAN Notices (ACM) *8*, 2 (1976), 25-35.
2. ASTRAHAN, M. M., ET AL. System R: Relational approach to database management. *ACM Trans. on Database Systems 1*, 2 (June 1976), 97-137.
3. BOYCE, R., ET AL. Specifying queries as relational expessions: SQUARE. Rep. RJ 1291, IBM Res. Lab., San Jose, Calif., Oct. 1973.
4. CHAMBERLIN, D., AND BOYCE, R. SEQUEL: A structured English query language. Proc. 1974 ACM-SIGMOD Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974, pp. 249-264.
5. CHAMBERLIN, D., GRAY, J.N., AND TRAIGER, I.L. Views, authorization and locking in a relational data base system. Proc. AFIPS 1975 NCC, Vol. 44, AFIPS Press, Montvale, N.J., May 1975, pp. 425-430.
6. Comm. on Data Systems Languages. CODASYL Data Base Task Group Rep., ACM, New York, 1971.
7. CODD, E.F. A relational model of data for large shared data banks. *Comm. ACM 13*, 6 (June 1970), 377-387.
8. 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, Calif., Nov. 1971, pp. 35-68.
9. CODD, E.F. Relational completeness of data base sublanguages. Courant Computer Science Symp. 6, May 1971, Prentice-Hall, Englewood Cliffs, N.J., pp. 65-90.

10. CODD, E.F., AND DATE, C.J. Interactive support for non-programmers, the relational and network approaches. Proc. 1974 ACM-SIGMOD Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974.
11. DATE, C.J., AND CODD, E.F. The relational and network approaches: Comparison of the application programming interfaces. Proc. 1974 ACM-SIGMOD Workshop on Data Description, Access and Control, Vol. II, Ann Arbor, Mich., May 1974, pp. 85-113.
12. GRAY, J.N., LORIE, R.A., and PUTZOLU, G.R. Granularity of Locks in a Shared Data Base. Proc. Int. Conf. of Very Large Data Bases, Framingham, Mass., Sept. 1975, pp. 428-451. (Available from ACM, New York.)
13. GO, A., STONEBRAKER, M., AND WILLIAMS, C. An approach to implementing a geo-data system. Proc. ACM SIGGRAPH/SIGMOD Conf. for Data Bases in Interactive Design, Waterloo, Ont., Canada, Sept. 1975, pp. 67-77.
14. GOTTLIEB, D., ET AL. A classification of compression methods and their usefulness in a large data processing center. Proc. AFIPS 1975 NCC, Vol. 44, AFIPS Press, Montvale, N.J., May 1975, pp. 453-458.
15. HELD, G.D., STONEBRAKER, M., AND WONG, E. INGRES—A relational data base management system. Proc. AFIPS 1975 NCC, Vol. 44, AFIPS Press, Montvale, N.J., 1975, pp. 409-416.
16. HELD, G.D. Storage Structures for Relational Data Base Management Systems. Ph.D. Th., Dep. of Electrical Eng. and Computer Science, U. of California, Berkeley, Calif., 1975.
17. HELD, G., AND STONEBRAKER, M. *B*-trees re-examined. Submitted to a technical journal.
18. IBM CORP. OS ISAM logic. GY28-6618, IBM Corp., White Plains, N.Y., 1966.
19. JOHNSON, S.C. YACC, yet another compiler-compiler. UNIX Programmer's Manual, Bell Telephone Labs, Murray Hill, N.J., July 1974.
20. MCDONALD, N., AND STONEBRAKER, M. Cupid—The friendly query language. Proc. ACM-Pacific-75, San Francisco, Calif., April 1975, pp. 127-131.
21. MCDONALD, N. CUPID: A graphics oriented facility for support of non-programmer interactions with a data base. Ph.D. Th., Dep. of Electrical Eng. and Computer Science, U. of California, Berkeley, Calif., 1975.
22. RITCHIE, D.M., AND THOMPSON, K. The UNIX Time-sharing system. *Comm. ACM 17*, 7 (July 1974), 365-375.
23. SCHOENBERG, I. Implementation of integrity constraints in the relational data base management system, INGRES. M.S. Th., Dep. of Electrical Eng. and Computer Science, U. of California, Berkeley, Calif., 1975.
24. STONEBRAKER, M. A functional view of data independence. Proc. 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974.
25. STONEBRAKER, M., AND WONG, E. Access control in a relational data base management system by query modification. Proc. 1974 ACM Nat. Conf., San Diego, Calif., Nov. 1974, pp. 180-187.
26. STONEBRAKER, M. High level integrity assurance in relational data base systems. ERI Mem. No. M473, Electronics Res. Lab., U. of California, Berkeley, Calif., Aug. 1974.
27. STONEBRAKER, M. Implementation of integrity constraints and views by query modification. Proc. 1975 SIGMOD Workshop on Management of Data, San Jose, Calif., May 1975, pp. 65-78.
28. STONEBRAKER, M., AND RUBINSTEIN, P. The INGRES protection system. Proc. 1976 ACM National Conf., Houston, Tex., Oct. 1976 (to appear).
29. TSICHRITZIS, D. A network framework for relational implementation. Rep. CSRG-51, Computer Systems Res. Group, U. of Toronto, Toronto, Ont., Canada, Feb. 1975.
30. WONG, E., AND YOUSSEFI, K. Decomposition—A strategy for query processing. *ACM Trans. on Database Systems 1*, 3 (Sept. 1976), 223-241 (this issue).
31. ZOOK, W., ET AL. INGRES—Reference manual, 5. ERL Mem. No. M585, Electronics Res. Lab., U. of California, Berkeley, Calif., April 1976.

Received January 1976; revised April 1976