# Retrospection on a Database System

MICHAEL STONEBRAKER
University of California at Berkeley

This paper describes the implementation history of the INGRES database system. It focuses on mistakes that were made in progress rather than on eventual corrections. Some attention is also given to the role of structured design in a database system implementation and to the problem of supporting nontrivial users. Lastly, miscellaneous impressions of UNIX, the PDP-11, and data models are given.

Key Words and Phrases: relational databases, nonprocedural languages, recovery, concurrency, protection, integrity
CR Categories: 3.50, 3.70, 4.22, 4.33, 4.34

## 1. INTRODUCTION

This paper was written in response to several requests to know what really happened in the INGRES database management system project [22] and why. To the extent that it contains practical wisdom for other implementation projects, it serves its purpose. To the extent that it is a self-righteous defense of the existing design, the author apologizes in advance.

It may be premature to write such a document, since INGRES has only been fully operational for three years and user experience is still somewhat limited. Hence the ultimate jury, real users, has not yet made a full report. The reason for reporting now is that we have reached a turning point. Until late 1978, the goal was to make INGRES "really work," i.e., efficiently, reliably, and without surprises (bugs) for users. There are now only marginal returns to pursuing that goal. Consequently, the project is taking new directions, which are discussed below.

This paper is organized as follows. In Section 2 we trace the history of the project through its various phases and highlight the more significant events that took place. Then, in Section 3, we discuss several lessons that we had to learn the hard way. Section 4 takes a critical look at the current design of INGRES and


The INGRES project was sponsored by the U.S. Air Force Office of Scientific Research under Grant 78-3596, the U.S. Army Research Office under Grant DAAG-29-G-0245, the Naval Electronics Systems Command under Contract N00039-78-G-0013, and the National Science Foundation under Grant MCS75-03839-AD1.
Author's address: Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720.

discusses some of the mistakes. Next, Section 5 consists of an assortment of random comments. Lastly, Section 6 outlines the future plans of the project.

## 2. HISTORY

The project can be roughly decomposed into three periods: (1) the early times—March 1973–June 1974; (2) the first implementation—June 1974–September 1975; (3) making it really work—September 1975–present. We discuss each period in turn.

### 2.1 The Early Times

The project began in 1973 when Eugene Wong and I agreed to read and discuss literature relating to relational databases. From the beginning we were both enthusiastic about an implementation. It did not faze either one of us that we possessed no experience whatsoever in leading a nontrivial implementation effort. In fact, neither of us had ever written a sizable computer program.

Our first task was to find a suitable machine environment for an implementation. It quickly become clear that no machine to which we had access was appropriate for an interactive database system. Through various mechanisms (mainly engineered by Eugene Wong and Pravin Varaiya) we obtained about $90,000 for hardware. The liability that we obtained was a commitment to write a geodata system for the Urban Economics Group led by Pravin Varaiya and Roland Artle.

Our major concerns in selecting hardware were in obtaining large (50 or 100 megabytes at the time) disks and a decent software environment. After studying the UNIX [18], I was convinced that we should use UNIX and buy whatever hardware we could afford to make it run. We placed a hardware order in February of 1974 and had a system in September of the same year.

We decided to offer a seminar running from September 1973 to June 1974 in which a design would be pursued. Somewhat symbiotically the seminar split into two groups: One group, led by Gene, would plan the user language; the other group, led by me, would plan the support system. The language group converged quickly on the retrieval portion of the data sublanguage QUEL. It was loosely based on DSL/Alpha [5] but had no notion of quantifiers.

As soon as UNIX was chosen, my group laid out the system catalogs (data dictionary) and the access method interface. Initially, we considered a nonrelational structure for the catalogs, as that would make them somewhat more efficient. However it quickly became clear that providing a specialized access facility for the system catalogs involved code duplication and would ruin the possibility of using QUEL to query the system catalogs. The latter feature would, in essence, provide a data dictionary system for free. Hence the system catalogs became simply more relational data for the system to manage.

An idea from the very start had been to have several implementations of the access method interface. Each would have the same calling conventions for simplicity and would function interchangeably. We were committed to the relational principle that users see nothing of the underlying storage structure. Hence no provisions were made to allow a user to access a lower level of the system (as is done in some other database systems).

During the winter of 1974 a lot of effort went into the tactics we would use for "solving" QUEL commands (query processing). The notion of tuple substitution [25] as a strategy for decomposing QUEL commands into simpler commands in QUEL itself was developed at this time. This notion of decomposition strongly influenced the resulting design. For example, having a level in the system that corresponded to the "one-variable query processor" occurred because decomposition required it.

In summary, the salient features of INGRES at the time were

(1) QUEL retrieval was defined;
(2) an integrated data dictionary was proposed;
(3) multiple implementations of the access methods were suggested;
(4) a "pure" relational system was agreed on;
(5) decomposition was developed.

This first period ended with the delivery in June 1974 of a PDP-11, which could be used on an interim basis for code development. Hence we could begin inplementing before our own machine arrived. The project was organized as a chief programmer team of four persons under the direction of Gerry Held. This same organizational structure remains today.

### 2.2 The First Implementation

We expected to exploit the natural parallelism which multiple UNIX processes allow. Hence decomposition would be a process to run in parallel with the one-variable query processor (OVQP). The utilities (e.g., to create relations, destroy them, and modify their storage structure) would be several overlays, but nobody was exactly sure where they would go. By this time we had decided to take protection seriously and realized that a database administrator (DBA) was an appropriate concept. He or she would own all the physical UNIX files in which relations for a given database were stored. In addition, the INGRES object code would use the "set user id" facility of UNIX so that it would run on behalf of any user with an effective user id of the DBA. This was the only way we could see to guarantee that nobody (except the DBA) could touch a database except by executing INGRES. Any less restrictive scheme would allow tampering with the database by other programs, which we thought undesirable.

Because the terminal monitor allowed the user to edit files directly, we had to protect the rest of INGRES from it. Hence it had to be a separate process. The notion of query modification for protection, integrity control, and views was developed during this time. It would be implemented with the parser, but no thought was given to the form of this module. During the summer of 1974 the process structure changed several times. Moreover, no one could coherently check any code because everyone needed the access methods as part of his code, and they did not work yet.

About this time another version of QUEL, which included updates and more general aggregates, was developed. This version survives today except for the keyword syntax, which was changed in early 1975.

By the end of the summer we had some access method code, some routines to access the data dictionary (to create and destroy relations, for example), and a

terminal monitor, along with pieces of DECOMP and OVQP. In September the department arranged to invite Ken Thompson (the creator of UNIX in conjunction with Dennis Ritchie) to Berkeley for a two-week visit. Ken was instrumental in getting UNIX to run on the INGRES machine and introduced us to YACC [14] as a parser generator.

In January of 1975 we invited Ted Codd to come to Berkeley in early March to see a demonstration of INGRES. The final two weeks before his visit everyone worked night and day so that we would have something to show him. What we demonstrated was a very "buggy" system with the following characteristics:

(1) The access methods "sort of" worked. Retrieves worked on all five implementations of the access methods (heap, hash, compressed hash, index, and compressed index). However, only heaps could be updated without fear of disaster.
(2) Decomposition was implemented by brute force.
(3) A primitive database load program existed, but few other services existed.
(4) All the messy interprocess problems had been ignored. For example, there was no way to reset INGRES so that it would stop executing the current command and be ready to do something new. Instead "reset" simply killed all of the INGRES processes and returned the user to the operating system command language interpreter.
(5) There were many bugs. For example, Boolean operators sometimes worked incorrectly. The average function applied to a relation with no tuples produced a weird response, etc.

At this point it became clear that the punctuation-oriented syntax for QUEL was horrible, and it was scrapped in favor of a keyword-oriented approach. The designers of SEQUEL [4] saw this important point sooner than we did. This was the last significant change made to the user language.

During this period we spent a lot of time discussing the pros and cons of dynamic directory facilities (e.g., B-treelike structures) and static directories (e.g., ISAM). The basic issue was whether the index levels in a keyed sequential access method were read-only or not. At the time we opted for static directories and wrote the paper "B-Trees Re-examined" [13]. This is one of the mistakes discussed in Section 4.

Lastly, it became clear that we needed a coupling to a host language. Moreover, C was the only possible candidate, since it alone allowed interprocess communication; a fact essential for INGRES operation. As a result, we began work on a preprocessor EQUEL [1], to allow convenient access to INGRES from C.

The end of this initial implementation period occurred when we acquired a user. Through Ken Thompson, to whom a tape of an early system had been sent, and through a group at Bell Laboratories in Holmdel, Dan Gielan of New York Telephone Co. become interested in using our system. After using our machine for a trial period, he obtained his own and set about tailoring INGRES to his environment and fixing its flaws (many bugs, bad performance, no concurrency control, no recovery, shaky physical protection, EQUEL barely usable). In a sense, during the next year he was duplicating much of the effort at Berkeley, and the two systems quickly and radically diverged.

The following issues were resolved during this period:

(1) QUEL syntax for updates was specified;
(2) the final syntax and semantics of QUEL were defined;
(3) protection was figured out;
(4) EQUEL was designed;
(5) concurrency control and recovery loomed on the horizon as big issues, and initial discussions on these subjects were started.

### 2.3 Making It Really Work

The current phase of INGRES development began during the latter part of 1975. At this time the system "more or less" worked. There were lots of bugs, and it was increasingly difficult to get them out. The system had performance problems due to convoluted and inefficient code everywhere. The code was also in bad shape. It had been constructed haphazardly by several people, not all of whom were still with the project. Each had his own coding style, way of naming variables, and library of common routines. In short, the system was unmaintainable.

The objective of the current phase was to make the system efficient, reliable, and *maintainable*. At the time we did not realize that this amounted to a total rewrite. We began to operate with more so-called "controls." No longer was there arbitrary tampering with the "current" copy of the code; rudimentary testing procedures were constructed, and rigid coding conventions were enforced. We began to operate more like a production software house and less like a freewheeling, unstructured operation.

During the current phase, concurrency control and recovery were seriously addressed. We took a long time to decide whether to take concurrency control seriously and write a sophisticated locking subsystem (such as the one in System R [8, 9]), or to do a quick and dirty subsystem using either coarse physical locks (say on files or collections of files) or predicate locks [7]. We also gave considerable thought to the size of a transaction. Should it be larger than one QUEL statement? If so, the simple strategy of demanding all needed resources in advance and avoiding deadlock was not possible.

Eventually it was decided to base the transaction size largely on simplicity. Once one QUEL statement was selected as the atomic operation for concurrency control and recovery, our hunch was that coarse physical locking would be best. This was later verified by simulation experiments [16, 17].

Recovery code was postponed as long as possible because it involved major changes to the utilities. All QUEL statements went through a "deferred update" facility, which made recovery from soft crashes (i.e., the disk remains intact) easy if a QUEL statement was being executed. The more difficult problem was to survive crashes while the utilities were running. Each utility performed its own manipulation of the system catalogs in addition to other functions. Leaving the system catalogs in a consistent state required being able to back up or run forward each command. The basic idea was to create an algorithm which would pass the system catalogs once (or at most twice), find all the inconsistencies regardless of what commands were running, and take appropriate action. Creating such a program required ironclad protocols on how the utilities were to manipulate the

system catalogs. Installing such protocols was a lot of work, most of it in the utilities, which by this time everyone regarded as boring code in enormous volume.

The parser had finally become so top-heavy from patches that it was rewritten from scratch. Decomposition was improved, and the system became progressively faster. In addition, the system was instrumented (no performance hooks were built in from the start). As a result we caught several serious performance botches. Elaborate tracing facilities were retrofitted to allow a decent debugging environment. In short, the entire system was rewritten.

During this time we also started to support a user community. There are currently some 100 users—all requesting better documentation, more features, and better performance. These became a serious time drain on the project.

Some of our early users appeared to be contemplating selling our software. We had taken no initial precautions to safeguard our rights to the code. It became necessary to prepare a license form and to pull everyone's lawyers into the act. This became a headache that could not easily be deflected, but which made technically supporting users look easy by comparison.

## 3. LESSONS

In this section some of the lessons that were learned from the INGRES project are discussed.

### 3.1 Goals

Our goals expanded several times (always when we were in danger of achieving the previous collection). Thus we added features which had not been thought about in the initial design (such as concurrency control and recovery) and began worrying about distributed databases (which had *never* been even talked about earlier). The effect of this goal expansion was to force us to rewrite a lot of INGRES, in some cases more than once.

### 3.2 Structured Design

The current wave of structured programming enthusiasts suggests the following implementation plan. Starting with the overall problem, one successively refines it until one has a tree structure of subproblems. Each level in such a tree serves as a "virtual machine" and hides its internal details from higher level machines. We encountered several problems in attempting to follow this seemingly sound advice. We discuss four of them.

(a) Use of structured design presumes that one knows what he is doing from the outset. There were many times we were confused with regard to how to proceed. In all cases we chose to do *something* as opposed to doing *nothing*, feeling that this was the most appropriate way to discover what we should have done. This philosophy caused several virtual machines to be dead wrong. Whenever this happened, a lot of redesign was inevitable. One example is the access method interface. This level was designed before it was completely understood how optimization concerning restricting scans of relations would be handled. It turned out that the interface chosen initially was ultimately not what we needed.

(b) We have had to contend with a 64K address space limitation. Initially we did not have a good understanding of how large various modules would be. On more than one occasion we ran out of space in a process which forced us into the unpleasant task of restructuring the code on space considerations alone. Moreover, since interprocess communication is not fast, we could not always structure code in the "natural" way because of performance problems.
(c) There was a strong temptation not to think out all of the details in advance. Because the design leaders had many other responsibilities, we often operated in a mode of "plan the general strategy and rough out the attack." In the subsequent detailed design, flaws would often be uncovered which we had not thought of, and corrective action would have to be taken. Often, major redesigns were the result.
(d) It was sometimes necessary to violate the information hiding of the virtual machines for performance reasons. For example, there is a utility which loads indexed sequential (ISAM-like) files and builds the directory structure. It is not reasonable to have the utility create an empty file and then add records one at a time through the access method. This strategy would result in a directory structure with unacceptable performance because of bad balance. Rather, one must sort the records, then physically lay them out on the disk, and then, as a final step, build the directory. Hence the program which loads ISAM files must know the physical structure of the ISAM access method. When this structure changed (and it did several times), the ISAM loader had to be changed.

All of these problems created a virtually constant rewrite/maintenance job of huge magnitude. In four years there were between two and five incarnations of all pieces of the system. Roughly speaking, we have rewritten a major portion of the system each year since the project began. Only now is code beginning to have a longer lifetime.

Earlier, there was hesitation on the part of the implementors to document code because it might have a short lifetime. Therefore documentation was almost nonexistent until recently.

### 3.3 Coding Conventions

To learn the necessity of this task was a very important lesson to us. As mentioned earlier, the equivalent of one total rewrite resulted from our initial failure in this area. We found that pieces of code which had a nontrivial lifetime were unmaintainable except by the original writer. Also, every time we gave someone responsibility for a new module, it would be rewritten according to the individual's personal standards (allegedly to clean up the other person's bad habits). This process never converges, and only coding conventions stop it.

### 3.4 User Support

There are lessons which we have learned about users in three areas.

3.4.1 *Serious Users.* There have been a few serious users (5–10). All are extremely bold and forward-looking people who have exercised our system extensively before committing themselves to use it. All of these users first chose

UNIX (which says something about their not being a random sample of users) and then obtained INGRES.

Most have made modifications to personalize INGRES to their needs, viewed us as a collection of goofy academicians, and been pretty skeptical that our code was any good. All have been very concerned about support, future enhancements, and how much longer our research grants would last.

All have developed end-user facilities using EQUEL and given us a substantial wish list of features. The following list is typical:

(1) The system is too slow (especially for trivial interactions).
(2) The system is too slow for very large databases (whatever this means).
(3) Protection, integrity constraints, and concurrency control are missing (true for earlier versions).
(4) The EQUEL interface is not particularly friendly.
(5) The system should have partial string-matching capabilities, a data type of "bit," and a macro facility. (The wish list of such features is almost unbounded.)

Surprisingly, nobody has ever complained about the crash recovery facilities. Also, a concurrency control scheme consisting of locking the whole database would be an acceptable alternative for most of our users.

The biggest problem that these users have faced is the problem of understanding some 500,000 bytes of source code, most of it free of documentation (other than comments in the code).

The merits of INGRES that most of these users claim, rest on the following:

(1) The system is easy to use after a minor amount of training. The "start-up" cost is much lower than for other systems.
(2) The high-level language allows applications to be constructed incredibly fast, as much as ten times faster than originally anticipated.

The short coding cycle allowed at least one user to utilize a novel approach to application design. The conventional approach is to construct a specification of the application by interacting with the end user. Then programmers go into their corner to implement the specifications. A long time later they emerge with a system, and the users respond that it is not really what they wanted. Then the rounds of retrofitting begin.

The novel approach was to do application specification and coding in parallel. In other words, the application designer interacted with end users to ascertain their needs and then coded what they wanted. In a few days he returned with a working prototype (which of course was not quite what they had in mind). Then the design cycle iterated. The important point is that end users were in the design loop and their needs were met in the design process. Only the ability to write database applications quickly and economically allowed this to happen.

3.4.2 *Casual Users.* There are about 90 more "casual" users. We hear less from these people. Most are universities who use the system in teaching and research applications. These users are less disgruntled with performance and unconcerned about support.

3.4.3 *Performance Decisions.* Users are not always able to make crucial performance decisions correctly. For example, the INGRES system catalogs are accessed very frequently and in a predictable way. There are clear instructions concerning how the system catalogs should be physically structured (they begin as heaps and should be hashed when their size becomes somewhat stable). Even so, some users fail to hash them appropriately. Of course, the system continues to run; it just gets slower and slower. We have finally removed this particular decision from the user's domain entirely. It makes me a believer in automatic database design (e.g., [11])!

## 4. FLAT OUT MISTAKES

In this section we discuss what we believe to be the major mistakes in the current implementation.

### 4.1 Interpreted Code

The current prototype interprets QUEL statements even when these statements come from a host language program. An interpreter is reasonable when executing ad hoc interactions. However the EQUEL interface processes interactions from a host language program as if they were ad hoc statements. Hence parsing and finding an execution strategy are done at run time, interaction by interaction.

The problem is that most interactions from host languages are simple and done repetitively. (For example, giving a 10 percent raise to a collection of employee names read in from a terminal amounts to a single parameterized update inside a WHILE statement.) The current prototype has a fixed overhead per interaction of about 400 milliseconds (400,000 instructions). Hence throughput for simple statements is limited by this fixed overhead to about 2.5 interactions per second. Parsing at compile time would reduce this fixed overhead somewhat.

At least as serious is the fact that the interpreter consumes a lot of space. The "working set" for an EQUEL program is about 150 kbytes plus the program. For systems with a limited amount of main memory this presents a terrible burden. A compiled EQUEL would take up much less space (at least for EQUEL programs with fewer than ten interactions per program). Moreover, a compiled EQUEL could run as fewer processes, saving us some interprocess communication overhead. This issue is further discussed in Section 4.3.

The interpreter was built with ad hoc interactions in mind. Only recently did we realize the importance of a programming language interface. Now we are slowly converting INGRES to be alternatively compiled and interpreted. We were clearly naive in this respect.

### 4.2 Validity Checking

This mistake is related to the previous one. When an interaction is received from a terminal or an application program, it is parsed at run time. Moreover (and at a very high cost), the system catalogs are interrogated to validate that the relation exists, that the domains exist, that the constants to which the domains are being compared are of the correct type or are converted correctly, etc. This costs perhaps 100 milliseconds of the 400-milliseconds fixed overhead, and no effort has been made to minimize its impact. This makes the "do nothing" overhead

high and, from a performance viewpoint, is the really expensive component of interpretation.

### 4.3 Process Problems

The "do nothing" overhead is greatly enlarged by our problems with a 16-bit address space. The current system runs as five processes (and the experimental system at Berkeley as six), and processing the "nothing" interaction requires that the flow of control go through eight processes. This necessitates formatting eight messages, calling the UNIX scheduler eight times, and invoking the interprocess message system (pipes) eight times. This generates about 150–175 milliseconds of the 400 milliseconds of fixed overhead.

In addition, code cannot be shared between processes. Hence the access methods must appear in every process. This causes wasted space and duplicated code. Moreover, some of the interprocess messages are the internal form of QUEL commands. As such, we require a routine to linearize a tree-structured object to pass through a pipe and the inverse of the routine to rebuild the tree in the recipient process. This is considerably more difficult than a procedure call passing as an argument a pointer to the tree. Again, the result is extra complexity, extra code, and lower performance.

Besides this performance problem, in the previous section it was noted that the process structure has changed several times because of space considerations. As a result, a considerable amount of energy has gone into designing new process structures, writing the code which correctly "spawns" the right run-time environment, and handling user interrupts correctly.

In retrospect, we had no idea how serious the performance problems associated with being forced to run multiple processes would be. It would have been clearly advantageous to choose a 32-bit machine for development; however, there was no affordable candidate to be obtained at the time we started. Also, perhaps we should have relaxed the 64K address limitation once we obtained a PDP-11/70 (which has a 128K limitation). This would have cut the number of processes somewhat. However, many of our 100 users have 11/34s or 11/40s and we were reluctant to cut them off. Lastly, we could have opted for less complexity in the code. However, to effectively cut the number of processes and the resulting overhead, the system would have to be reduced by at least a factor of 2. It is not clear that an interesting system could be written within such a constraint. The bottom line is that this has been an enormous problem, but one for which we see no obvious solution other than to buy a PDP-11/780 and correct the situation now that a 32-bit machine which can run our existing code is available.

### 4.4 Access Methods

Very early the decision was made not to write our own file system to get around UNIX performance (as System R elected to do for VM/370 [2]). Instead, we would simply build access methods on top of the existing file system.

The reasoning behind this decision was to avoid duplicating operating system functions. Also, exporting our code would have been more difficult if it contained its own file system. Lastly, we underestimated the severity of the performance degradation that the UNIX file system contributes to INGRES when it is

processing large queries. This topic is further discussed in [12]. In retrospect, we probably should have written our own file system.

The other problem with the access methods concerns whether they are I/O bound. Our initial assumption was that it would never take INGRES more than 30 milliseconds to process a 512-byte page. Since it takes UNIX about this long to fetch a page from the disk, INGRES would always be I/O bound for systems with a single-disk controller (the usual case for PDP-11 environments). Although INGRES is sometimes I/O bound, there are significant cases where it is CPU bound [12].

The following three situations are bad mistakes when INGRES is CPU bound:

(a) An entire 512-byte page is always searched even if one is looking only for one tuple (i.e., a hash bucket is a UNIX page).
(b) A tuple may be moved in main memory one more time than is strictly necessary.
(c) A whole tuple is manipulated, rather than just desired fields.

Although we have corrected points (b) and (c), point (a) is fundamental to our design and is a mistake.

### 4.5 Static Directories

INGRES currently supports an indexing access method with a directory structure which is built at load time and never modified thereafter. The arguments in favor of such a structure are presented in [13]. However, we would implement a dynamic directory (as in B-trees) if the decision were made again. Two considerations have influenced the change in our thinking.

The database administrator has the added burden of periodically rebuilding a static directory structure. Also, he can achieve better performance if he indicates to INGRES a good choice for how full to load data pages initially. In the previous section we indicated that database administrators often had trouble with performance decisions, and we now believe that they should be relieved of all possible choices. Dynamic directories do not require periodic maintenance.

The second fundamental problem with static directories is that buffer requirements are not predictable. In order to achieve good performance, INGRES buffers file system pages in user space when advantageous. However, when overflow pages are present in a static directory structure, INGRES should buffer all of them. Since address space is so limited, a fixed buffer size is used and performance degrades severely when it is not large enough to hold all overflow pages. On the other hand, dynamic directories have known (and nearly constant) buffering requirements.

### 4.6 Decomposition

Although decomposition [25] is an elegant way to process queries and is easy to implement and optimize, there is one important case which it cannot handle. For a two-variable query involving an equijoin, it is sometimes best to sort both relations on the join field and then merge the results to identify qualifying tuples [3]. Consequently, it would be desirable for us to add this as a tactic to apply when appropriate. This would require modifying the decomposition process to

look for a special case (which is not very hard) and, in addition, restructuring the INGRES process structure (since query processing is in two UNIX processes, and this would necessarily alter the interface between them). Again, the address space issue rears its ugly head!

### 4.7 Protection

It appears much cleaner to protect "views" as in [10] rather than base relations as in [19, 21]. It appears that sheer dogma on my part prevented us from correcting this.

### 4.8 Lawyers

I would be strongly tempted to put INGRES into the public domain and delete our interactions with all attorneys (ours and everyone else's). Whatever revenue the University of California derives from license fees may well not compensate for the extreme hassle which licensing has caused us. Great insecurity and our egos drove us to force others to recognize our legal position. This was probably a big mistake.

### 4.9 Usability

Insufficient attention has been paid to the INGRES user interface. We have learned much about "human factors" during the project and have corrected many of the botches. However, there are several which remain. Perhaps the most inconvenient is that updates are "silent." In other words, INGRES performs an update and then responds a "done." It never gives an indication of the tuples that were modified, added, or deleted (or even how many there were). This "feature" has been soundly criticized by almost everyone.

## 5. COMMENTS

This section contains a collection of comments about various things which do not fit easily into the earlier sections.

### 5.1 UNIX

As a program development tool, we feel that UNIX has few equals. We especially like the notion of the command processor; the notion of pipes; the ability to treat pipes, terminals, and files interchangeably; the ability to spawn subprocesses; and the ability to fork the command interpreter as a subprocess from within a user program. UNIX supports these features with a pleasing syntax, very few "surprises," and most unnecessary details (e.g., blocking factors for the file system) remain hidden.

The use of UNIX has certainly expedited our project immeasurably. Hence we would certainly choose it again as an operating system.

The problems which we have encountered with UNIX have almost all been associated with the fact that it was envisioned as a general-purpose time-sharing system for small machines and not as a support system for database applications.

Hence there is no concurrency control and no crash recovery for the file system. Also, the file system does not support large files (16 Mbytes is the current limit) and uses a small (512 bytes) page size. Moreover, the method used to map logical

pages to physical ones is not very efficient. In general, it appears that the performance of the file system for our application could be dramatically improved.

### 5.2 The PDP-11

Other than the address space problems with a PDP-11, I have only two other comments regarding the hardware. First, there is no notion of "undefined" as a value for numeric data types supported by the hardware. Allowing such a notion in INGRES would require taking some legal bit pattern and by fiat making it equal undefined. Then we would have to inspect every arithmetic operation to see if the chosen pattern happened inadvertently. This could be avoided by simple hardware support (such as found on CDC 6000 machines).

Second there is no machine instruction which can move a string in main memory. Consequently, data pages are moved in main memory one word at a time inside a loop. This is a source of considerable inefficiency.

### 5.3 Data Models

There has been a lot of debate over the efficiency of the various data models. In fact, a major criticism of the relational model has been its (alleged) inefficiency.

There are (at least) two ways to compare the performance of database systems.

(a) The overhead for small transactions. This is a reasonable measure of how many transactions per second can be done in a typical commerical environment.
(b) The cost of a given big query.

It should be evident that (a) has nothing to do with the data model used (at least in a PDP-11 environment). It is totally an issue of the cost of the operating system, system calls, environment switches, data validity costs, etc. In fact, if INGRES were a network-oriented system and ran as five processes, it would also execute 2.5 transactions per second.

The cost of a big query is somewhat data model dependent. However, even here this cost is extremely sensitive to the cost of a system call, the operating system decisions concerning buffering and scheduling, the cost of shuffling output around and formatting it for printing, and the extent to which clever tuning has been done. In addition, the design of a database management system is often very sensitive to the features (and quirks) of the operating system on which it is constructed. (At least INGRES is.) These are probably much more important in determining performance than what data model is used.

In summary, I would allege that a comparison of two systems using different data models would result primarily in a test of the underlying operating system and the implementation skill (or man-years allowed) of the designers and only secondarily in a test of the data models.

## 6. INGRES PROJECT PLANS

INGRES appears to be at least potentially commercially viable. However a commercial version would require, at least

(1) someone to market it;
(2) much better documentation;

(3) someone willing to guarantee maintenance (whether or not we do it, the University of California will not promise to fix bugs);
(4) a pile of boring utilities (e.g., a report generator, a tie into some communications facilities, and access to the system from languages other than C).

Even so, we would not have a good competitive position because UNIX is not supported and because no Cobol exists for UNIX.

There has been a clear decision on the part of the major participants not to create a commercial product (although that decision is often reexamined). On the other hand, the project cannot simply announce that it has accomplished its goals and close shop. Hence we have gone through a (sometimes painful) process of self-examination to decide "what next." Here are our current plans.

### 6.1 Distributed INGRES

We are well into designing a distributed database version of INGRES which will run on a network of PDP-11s. The idea here is to hide the details of location of data from the users and fool them into thinking that a large unified database system exists [6, 23].

### 6.2 A Distributed Database Machine

This is a variant on a distributed database system in which we attempt only to improve performance. It has points in common with "back end machines" and depends on customizing nodes to improve performance [24].

### 6.3 A New Database Programming Language

Obviously, starting with C and an existing database language QUEL and attempting to glue them together into a composite language is rather like interfacing an apple to a pancake. It would clearly be desirable to start from scratch and design a good language. Initial thoughts on this language are presented in [15].

### 6.4 A Data Entry Facility

An application designer must write EQUEL programs to support his customized interface. The portion of such programs that can be attributed to the database system has shrunk to near zero (by the high-level language facilities of QUEL). Hence we are left with transactions that have virtually no database code and are entirely what might be called "screen definition, formatting, and data entry." We are designing a facility to help in this area.

### 6.5 Improved Integrity Control

Currently, INGRES is not very smart in this area. Other than integrity constraints [20] (which do something but not as much as might be desired), we have no systematic means to assist users with integrity/validation problems. We are investigating what can be done in this area.

It is pretty clear that all of the above will require substantial changes in the current software. Hence we can remain busy for a seemingly arbitrary amount of time. This will clearly continue until we get tired or are again in danger of meeting our goals.

ACKNOWLEDGMENT

The INGRES project has been directed by Profs. Eugene Wong and Larry Rowe in addition to myself. The role of chief programmer has been filled by Gerald Held, Peter Kreps, Eric Allman, and Robert Epstein. The following persons worked on the project at various times; Richard Berman, Ken Birman, James Ford, Paula Hawthorn, Randy Katz, Nancy MacDonald, Marc Meyer, Daniel Ries, Peter Rubinstein, Polly Siegel, Michael Ubell, Nick Whyte, Carol Williams, John Woodfill, Karel Youseffi, and William Zook.

REFERENCES

1. ALLMAN, E., HELD, G., AND STONEBRAKER, M. Embedding a data manipulation language in a general purpose programming language. Proc. ACM SIGPLAN SIGMOD Conf. on Data Abstractions, Salt Lake City, Utah, March 1976, pp. 25–35.
2. ASTRAHAN, M.M., ET AL. System R: Relational approach to database management. *ACM Trans. Database Syst. 1*, 2 (June 1976), 97–137.
3. BLASGEN, M., AND ESWARAN, K. Storage and access in relational data base systems. *IBM Syst. J.* (Dec. 1977), 363–377.
4. CHAMBERLIN, D.D., AND BOYCE, R.F. SEQUEL: A structured English query language. Proc. ACM SIGFIDET Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974, pp. 249–264.
5. CODD, E.F. A database sublanguage founded on the relational calculus. Proc. ACM SIGFIDET Workshop on Data Description, Access and Control, San Diego, Calif., Nov. 1971, pp. 35–68.
6. EPSTEIN, R., STONEBRAKER, M., AND WONG, E. Query processing in a distributed data base system. Proc. ACM SIGMOD Conf. on Management of Data, Austin, Tex., May 1978, pp. 169–180.
7. ESWARAN, K.P., GRAY, J.N., LORIE, R.A., AND TRAIGER, I.L. The notions of consistency and predicate locks in a database system. *Comm. ACM 19*, 11 (Nov. 1976), 624–633.
8. GRAY, J.N., LORIE, R.A., PUTZOLU, G.R., AND TRAIGER, I.L. Granularity of locks and degrees of consistency in a shared data base. Res. Rep. RJ 1849, IBM Research Lab., San Jose, Calif., July 1976.
9. GRAY, J. Notes on data base operating systems. Res. Rep. RJ 2188, IBM Research Lab., San Jose, Calif., Feb. 1978.
10. GRIFFITHS, P.P., AND WADE, B.W. An authorization mechanism for a relational database system. *ACM Trans. Database Syst. 1*, 3 (Sept. 1976), 242–255.
11. HAMMER, M., AND CHAN, I. Index selection in a self adaptive data base system. Proc. ACM SIGMOD Conf. on Management of Data, Washington, D.C., June 1976, pp. 1–8.
12. HAWTHORN, P., AND STONEBRAKER, M. Use of technological advances to enhance data base management system performance. Memo No. 79-5, Electronics Res. Lab., U. of California, Berkeley, Calif., Jan. 1979.
13. HELD, G., AND STONEBRAKER, M. B-trees re-examined. *Comm. ACM 21*, 2 (Feb. 1978), 139–143.
14. JOHNSON, S. YACC—yet another compiler-compiler. Comptr. Sci. Tech. Rep. No. 32, Bell Telephone Laboratories, Murray Hill, N. J., July 1975.
15. PRENNER, C., AND ROWE, L. Programming languages for relational data base systems. Proc. Nat. Comptr. Conf., Anaheim, Calif., June 1978, pp. 849–855.
16. RIES, D.R., AND STONEBRAKER, M. Effects of locking granularity in a database management system. *ACM Trans. Database Syst. 2*, 3 (Sept. 1977), 233–246.
17. RIES, D.R., AND STONEBRAKER, M.R. Locking granularity revisted. *ACM Trans. Database Syst. 4*, 2 (June 1979), 210–227.
18. RITCHIE, D.M., AND THOMPSON, K. The UNIX time-sharing system. *Comm. ACM 17*, 7 (July 1974), 365–375.
19. STONEBRAKER, M.R., AND WONG, E. Access control in a relational data base management system by query modification. Proc. ACM Ann. Conf., San Diego, Calif., Nov. 1974, pp. 180–187.
20. STONEBRAKER, M. Implementation of integrity constraints and views by query modification. Proc. ACM SIGMOD Conf. on Management of Data, San Jose, Calif., May 1975, pp. 65–78.

21. STONEBRAKER, M., AND RUBINSTEIN, P. The INGRES protection system. Proc. ACM Ann. Conf., Houston, Tex., Nov. 1976, pp. 80–84.
22. STONEBRAKER, M., WONG, E., AND KREPS, P. The design and implementation of INGRES. *ACM Trans. Database Syst. 1*, 3 (Sept. 1976), 189–222.
23. STONEBRAKER, M. Concurrency control, crash recovery and consistency of multiple copies of data in a distributed data base system. Proc. 3rd Berkeley Workshop on Distributed Data Bases and Computer Networks, San Francisco, Calif., Aug. 1978, pp. 235–258.
24. STONEBRAKER, M. MUFFIN: A distributed data base machine. Proc. First Int. Conf. on Distributed Computing Systems, Huntsville, Ala., Oct. 1979, pp. 459–469.
25. WONG, E., AND YOUSEFFI, K. Decomposition—a strategy for query processing. *ACM Trans. Database Syst. 1*, 3 (Sept. 1976), 223–241.

Received January 1979; revised September 1979; accepted September 1979