

PERFORMANCE ANALYSIS OF SEVERAL BACKEND DATABASE ARCHITECTURES

Robert Brian Hagmann

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California
Berkeley, California 94720

ABSTRACT

With the growing acceptance of database systems, the performance of these systems becomes increasingly more important. One way to gain performance is to off-load some of the functions of the database system to a backend computer. The problem is what functions should be off-loaded to maximize the benefits of distributed processing.

The approach in this research consists of constructing several variants of an existing relational database system, INGRES, that partition the database system software into two parts, and assigning these two parts to two computers connected by a local area network. For the purposes of this experiment, six different variants of the database software were constructed to test the six most interesting functional subdivisions. Each variant was then benchmarked using two different databases and query streams. The communication medium and software were also benchmarked to measure their contribution to the performance of each configuration.

Combining the database and network measurement results, various conclusions were reached about the viability of the configurations, the desirable properties of the communications mechanisms to be used, the operating system interface and overhead, and the performance of the database system. The variants to be preferred depend on the hardware technology, operating system features, database system internal structure, and network software overhead.

PERFORMANCE ANALYSIS OF
SEVERAL BACKEND DATABASE ARCHITECTURES

Copyright © 1983

Robert Brian Hagmann

Table of Contents

Chapter 1. Introduction	1
1.1 The Problem	1
1.2 Motivation	1
1.3 Approach	2
1.4 Functions of a Database System	2
1.5 Outline	2
Chapter 2. Survey of Backend Database Machines	3
2.1 Simple Disk Controllers	3
2.2 Complex Disk Controllers	3
2.3 Simple Backend with a Low Level Interface	4
2.3.1 Software AG's ADABAS Data Base Machine	4
2.3.2 Experimental Data Management System	4
2.3.3 Copernique MIX's Navigational Interface	4
2.4 Backend Machines with a High Level Interface	4
2.4.1 Britton-Lee's Intelligent Database Machine	4
2.4.2 Intel's Database Processor	4
2.4.3 Copernique MIX's Relational Interface	4
2.4.4 ICL's Content Addressable File Store	5
2.4.5 DIRECT	5
2.4.6 Ohio State University's Data Base Computer	5
2.5 Conclusion	5
Chapter 3. Experiment Design	6
3.1 Objectives	6
3.2 The Experimental Approach	6
3.3 The Basis for the Experiment	7
3.4 A General Description of INGRES	9
3.5 The Configurations of INGRES	9
3.6 Implementation	16
3.7 Buffering and Non-volatile Storage	17
3.8 Databases Used in the Experiment	18
3.9 Network Benchmarks	18
3.10 Criteria for Comparison of the Configurations	18
3.11 Measurements and Instrumentation	18
Chapter 4. Experimental Results	20
4.1 Network Measurements	20
4.2 Database System Measurements	23
4.2.1 CPU Utilization Measurements	23
4.2.2 Disk I/O Measurements	29
4.2.3 Network Overhead Results	35
4.2.4 Summary Time Data	39
4.3 Page Buffering in the Backend	42
4.4 Non-volatile Storage	45
4.5 Granularity of Communication	48

4.6 Performance Statistics under UNIX	48
4.7 The INGRES Process ID	49
Chapter 5. Discussion	50
5.1 Configuration Comparisons	50
5.1.1 Ingres	50
5.1.2 Smart Disk	51
5.1.3 Access Methods	52
5.1.4 Inner Loop	52
5.1.5 Decomposition	52
5.1.6 Parser	52
5.2 Experiment Versus Modeling or Simulation	54
Chapter 6. Conclusions and Future Work	54
6.1 Conclusions	55
6.2 Future Work	56
Bibliography	56

CHAPTER 1

Introduction

As databases grow in size and the use of them becomes more widespread, the load on a computer due to a database system becomes a prime performance problem. This thesis examines various methods for off-loading different functional parts of the database software to a second machine. The goal is to reduce the load on the first computer by distributing part of the database system to the second computer. For the remainder of this thesis, this additional machine will be called a backend computer, and the first computer will be called the frontend.

This dissertation investigates the performance impact of a database system's functional subdivision by constructing the proposed software configurations. Once the functional subdivisions were constructed, experiments were run to determine their performance. The software of a real database system is modified so that parts of it can run on different computers. A wide spectrum of functional subdivisions for the database software is examined.

This is an experimental thesis. An empirical approach was chosen because a database system is not a stand alone entity: it executes on physical hardware and (usually) runs under the control of an operating system. The interrelationships among hardware, operating system, and hardware are quite complex and can best be studied by observing an actual system. This work studies not only database system performance, but also the interactions of the database system with various hardware devices and operating system features.

1.1. The Problem

This thesis attempts to address the general problem of how should the software which implements a database system's functions be functionally subdivided between the frontend and backend. One way to view the two parts of the database system is to consider them as running on a pair of computers consisting of the frontend and the backend machine. However, the problem addressed in this work does not only arise in the case of a central computer assisted by a backend machine. It is also found in a distributed system consisting of a set of computers, possibly personal computers, that communicate with one or more database servers. In both cases, the database software must be functionally partitioned into two pieces: the code for the frontend(s) and the code for the backend (or database server).

In addition to this central problem, the dissertation addresses some issues concerned with the interrelations between a database system, the underlying hardware, and the operating system. In particular, some insight is gained into the benefits of using non-volatile storage and into the desirability of implementing low level protocol functions for communications in hardware. As far as the operating system is concerned, some data about the effectiveness of buffering and overhead are presented.

1.2. Motivation

The issue of database system performance has existed since the first of these systems were built. Several machines to either assist or perform the database function have been proposed, prototyped, and in some cases built. A survey of these machines is given in chapter two. Each of these machines has been built upon intuition: its creators have proposed a technical solution to what they perceived as the performance problem of database systems. A second way to approach the problem of database machine performance, the method used in this work, is the experimental way: with this method, each of several proposed functional divisions of the database system is built and tested before deciding the tasks to be assigned to the backend machine and how the system ought to be organized.

Database systems are hard to construct or run poorly since the functions they must perform often are not in harmony with those provided by the operating system or the underlying hardware. Examples of these discrepancies are double paging (the operating system pages the

database system's buffer pool), delayed writes (the operating system delays a file system write to gain efficiency), lack of fast stable storage (any data that must survive a crash or power failure must be written to disk), and no user critical sections (so that the database system's internal resource managers, such as the buffer pool and lock managers, can perform atomic operations) [Blas79] [Gold74] [Gray78] [Gray79] [Lind79] [Ston80] [Ston81] [Trai82]. Much of the problem stems from the fact that database systems are dependent on physical aspects of the hardware for crash recovery and performance. Operating systems attempt to hide certain features of the underlying hardware from the users. For unsophisticated programs this is probably a good decision, but for sophisticated programs, like database systems, this hiding can be harmful. Database systems are pliable in their memory needs, can often predict disk accesses, need to perform atomic updates to data structures in memory and on disk, and can be designed capitalizing on the knowledge of the physical characteristics of disks. Hence, in addition to studying the functional decomposition of a database system, this thesis was designed to investigate some of the important aspects of the interface between the database system and both the operating system and the hardware.

1.3. Approach

The question of how should the functions of a database system be subdivided was addressed by performing a series of experiments. An existing relational database system, INGRES [Ston76], was chosen as the system to be functionally distributed. The one selected was the research version of INGRES, that runs on the DEC VAX-11¹ series of computers under the Berkeley version of the UNIX² [Ritc78] [Joy81b] operating system. Criteria were established for subdividing the system, and six functional partitions of the database software were produced. The software of INGRES was then modified to make each of the functional subdivisions, to be called a configuration in the sequel, a running piece of software. The two dedicated computers were connected by a local area network. Two databases and two sequences of queries were obtained. Each of the six configurations was then benchmarked on the two-machine system to get performance data. In addition, the network was benchmarked in order to distinguish the contributions of database performance from those of network performance.

This methodology was followed to achieve the closest possible approximation to a real system based on each functional division considered in this investigation. It was felt that a less ambitious attempt to address this problem would not produce sufficiently accurate data. The experimental approach using real software, real databases, and real computers provided the unique opportunity to observe how an actual system implementing the various software configurations would run.

1.4. Functions of a Database System

The basic functions of a database system are that it provides for the storage, access, insertion, deletion, and modification of data. It also provides for data protection (preventing access to some data by some users), integrity (making sure the data satisfies a consistency constraint), transaction management, concurrency control, and crash recovery. Optionally, it may also provide a high level query language to ease the use of the system.

1.5. Outline

This thesis has five more chapters. The next chapter contains a summary of some of the relevant work in the field of database machines. Chapter 3 describes the concepts on which the design of the experiment was based. Chapter 4 presents and discusses the results of the various benchmarking sessions. Chapter five analyzes by configuration the data presented in the previous chapter. The last chapter presents the conclusions and discusses avenues for future work.

¹DEC and VAX are trademarks of Digital Equipment Corporation

²UNIX is a trademark of Bell Laboratories

CHAPTER 2

Survey of Backend Database Machines

It is normal practice in a thesis to survey the related work in the field. Although there is much literature on database systems in general ([Date75] and [Ullm80] for example) and on backend database machines (see below), we were unable to find any published papers that applied to the study of database systems a methodology similar to that used in this dissertation. The purpose of this chapter is to discuss several backend database machines that have been proposed. By no means are all papers in this field discussed in the text of this chapter, or even referenced in the bibliography. For example, [Aror81], [Banc80], [Doga80], [Kiy81], [Kung80], [Lin76], [Mari75], [Shaw82], [Shib82], [Su75], and [Wah80] describe backend database systems not discussed here. See also [Date83] for additional references. The machines discussed in this chapter are commercial products, or the focus of a research project at some university. The discussion is not about how the systems work, but is rather concerned with how database system functions are divided between their hardware components.

This chapter is divided into four sections. The first section is for simple disk controllers that are used in typical database systems. The next section covers complex disk controllers. These are systems where the controller has some decision making capabilities, but has no knowledge of the semantics of the data on the disk. The third section deals with backends where the interface is at a fairly low level. The backend does, however, have some model for the semantics of the data. The last section covers backends whose interface is at a high level. These backends have a large portion (or all) of the database system is moved to the backend.

2.1. Simple Disk Controllers

In common use today for the storage of moderately large amounts of information are magnetic disks. Although there is a quite large performance spectrum for these devices, a typical "hard" disk has an average access time of 30 milliseconds, and transmits data at a rate of about 1 megabyte per second [Ampe82].

A typical disk controller handles physical positioning to the disk arm, converts between the digital and the analog forms of the data, and performs direct memory access (DMA) to transfer the data to/from the disk and memory [Emul80]. There may be some buffering of data in the controller, but this is typically a small number of sectors. The controller does not do scheduling of the data transfers.

2.2. Complex Disk Controllers

There are two types of hardware products that fit into the category of complex disk controllers. First, there are disk controllers that can be directly attached to the frontend by means of a channel or by DMA, and provide buffering and other services to increase performance. An example is the IBM 3380 Model 13 disk controller for the IBM 3380 Direct Access Storage (a disk) [IBM81]. The controller has up to 8 megabytes of magnetic bubble buffer storage. The prime purpose of this buffering is to assist with channel/controller/string contention problems, and not to help a database system perform its functions. As such, this controller would not really qualify as a database system oriented complex disk controller. However, this suite of hardware is very close to that needed by a complex disk backend (the *smart disk* configuration described in section 3.5). Only microcode and some hopefully minor engineering changes would be necessary to make it more useful to a database system.

A second type of complex disk controller is a file server. The frontends access the file server over a local area network. The file server has simple disk controllers, as described in the previous subsection, attached to it. The server can buffer, pre-fetch data, schedule the disk arms, and provide some sort of transaction abstraction for the frontend. Descriptions of some examples of file servers, not all of which have all the above features, can be found in [Swin79], [Dion80], [Frid81],

and [Mitc82].

2.3. Backend Machines with a Low Level Interface

2.3.1. Software AG's ADABAS Data Base Machine

The ADABAS Data Base Machine [Soft] [Soft81] supports a network model database system. The types of calls that pass over the interface between the frontend and backend are the Data Manipulation Language (DML) commands. Examples of these commands are FIND (and its variants), MODIFY, GET, INSERT, and REMOVE. The database is offloaded as much as possible to the backend given the restrictions of the database model. The backend (an Externally Supported Processor) sold by Software AG is functionally identical to members of the IBM 370 series. It is connected to an IBM 370 series frontend using a channel-to-channel communications system (CTCS).

2.3.2. Experimental Data Management System

A backend database machine that is part of a CODASYL [CODA71] database system is the Experimental Data Management System (XDMS) [Cana74]. The interface of XDMS is at the DML command level. The frontend for the prototype was the UNIVAC 1108, and the backend was a Digital Scientific META-4.

2.3.3. Copernique MIX's Navigational Interface

MIX is a commercial backend database machine being built by Copernique [Armi81]. There are two different interfaces to the MIX backend: the navigational and the relational interface. (A navigational database system is either a network or hierarchical database system. It is called navigational because the the interface to the database is such that the user directs (navigates) the system through the data.) When using the navigational interface, MIX implements a CODASYL style data model. Using either interface, MIX performs the following functions: data retrieval/insert/update, data dictionary, transaction control, locking, and crash recovery.

2.4. Backend Machines with a High Level Interface

2.4.1. Britton-Lee's Intelligent Database Machine

The Intelligent Database Machine (IDM) [Epst80] [Ubel82] implements almost an entire relational database system in the backend machine. The only part of the database system that remains in the frontend is the parser for queries. The frontend requests action by the backend by transmitting a parsed query tree to the backend. Protection, query planning, validity checking, logging and crash recovery, among other functions, are all performed in the backend. The backend is a 16-bit microprocessor chip (Zilog Z8000), possibly assisted by a "Database Accelerator". The "operating system" in the backend is quite small and is oriented toward supporting database functions.

2.4.2. Intel's Database Processor

The Intel Database Processor (iDBP) [Inte82] is a backend machine capable of supporting relational as well as navigational database systems. The interface to the backend is via encoded iDBP commands. Examples of commands are JOIN, SELECT, and PROJECT (all subtypes of the DEFINE VIEW command), as well as DELETE, FIND, MODIFY and FETCH. To use this type of backend, a query parser executing in the frontend must compile queries into a series of high level commands for the backend.

2.4.3. Copernique MIX's Relational Interface

MIX, described above, also has a relational interface. The database system calls take the form of queries in SEQUEL [Cham74]. All database operations, including parsing, are performed in the backend.

2.4.4. ICL's Content Addressable File Store

International Computers Limited's Content Addressable File Store (CAFS³) is an example of a search processor [Babb79]. The backend is essentially a smart disk controller that has special key matching hardware. It also has a "bit array store" that is used to mark tuples. The use of the bit array store enable CAFS to perform simple relational queries in only a few commands from the frontend. Up to twelve disk tracks from a single disk can be handled concurrently by the backend.

2.4.5. DIRECT

DIRECT is a project aimed at building a relational database backend at the University of Wisconsin [DeWi78] [DeWi79] [Bora81]. The frontend, a PDP⁴ 11/40 or a VAX 11/750, compiles queries and does query planning. The backend consists of a controller running on a PDP 11/40 and a set of "query processors" executing on LSI 11/23's. The planner running in the frontend decomposes a query into a series of one and two variable queries all of which together are called a "query packet". Examples of the types of these smaller queries are RESTRICT, JOIN, PROJECT, and INSERT. The controller receives a query packet and plans the query execution using as many of the LSI 11/23's as possible.

2.4.6. Ohio State University's Data Base Computer

The Data Base Computer (DBC) is a backend database machine at Ohio State University that implements the attribute-based data model [Bane78] [Hsia81]. The interface to the backend makes use of a high-level query language. The operations the backend performs are security enforcement and the insertion, retrieval, and deletion of records. The DBC does not run a typical operating system. It employs a variety of specialized hardware components to implement database system functions.

2.5. Conclusion

This chapter has presented a survey of some of the literature about database machines. The next chapter describes the experiment itself.

³CAFS is a trademark of International Computers Limited

⁴PDP is a trademark of Digital Equipment Corporation

CHAPTER 3

Experiment Design

3.1. Objectives

The prime objective of this thesis is to investigate the performance effects of various subdivisions of a database system's functions between a frontend and a backend computer. Not only are the raw values of the performance indices of each configuration of interest, but also the factors that influence these values. For example, one might conjecture that the performance of a configuration is completely dominated by the network overhead, but the validity of this conjecture will not be established unless the network overhead is measured together with the global metrics of the system performance.

A database system interfaces directly with both the operating system and the underlying hardware. A second objective of this thesis is to understand and quantify some of the interrelationships among these three components of a system. In particular, the thesis investigates the causes of disk I/O operations, some buffering issues, and the use of non-volatile storage to avoid or delay disk writes.

3.2. The Experimental Approach

At least four methods can be employed to answer the principle question examined in this thesis: analytical modeling, simulation, careful measurement of a database system running on a single computer, and construction and measurement of a multiple computer database system. Each of these approaches represents a well respected method of investigation. However, both analytical modeling and simulation are critically dependent on the assumptions the evaluator makes about what is important to model. During the selection process for an approach it was felt that in this study it would be impossible to make sufficiently good assumptions to justify a modeling method (see section 5.2). The careful measurement of a single computer system could be effective if the functions that would in reality be performed by each of the separate computers could easily be identified and measured. This approach was rejected for two reasons. First, it was felt it would be difficult to get good results because of the low clock resolution on the available machines (which were DEC VAX-11's), and of the inaccuracies that characterize the instrumentation available in the UNIX operating system. The second reason was that the difficulty of really building a distributed system appeared to be only slightly greater than that of accurately measuring a single computer. Furthermore, the distributed system solution had the advantage that errors made in deciding how system functions should be partitioned would become bugs in the implementation, while they would go undetected in the measurement of a single system. These errors could take two forms. First, there could be simple errors in the analysis of the functional subdivision. An example of this is assigning a backend function to the frontend. Second, there could be errors due to incorrectly assigning the use of functions that were used in both the frontend and the backend. An example of this would be to mislocate the temporary files for the terminal monitor so that unneeded communication would be performed.

Hence, the fourth approach was adopted to get the best possible results. This approach has the advantage over analytical modeling and simulation of not requiring any simplifying (hence probably distorting) assumptions. Over a single-computer database system, it has the advantage that the system has really to work: its running properly cannot mask possible errors in the partitioning of its functions.

Once a configuration was identified, the software of the database system was examined to determine exactly which functions would be migrated to the backend. Sometimes these functions were performed by a complete process or a set of modules in the original database system. In these cases, the mechanisms used by the processes or modules to communicate with that part of the system that would reside in the frontend machine could be simply converted to allow them to

communicate over a network connection. In the rest of the configurations, software was constructed to perform the necessary interfacing functions. The implementation of this interface usually took the form of a remote procedure call. That is, the arguments and return values were specified by copy semantics (not by pointers) and the call was synchronous (blocking). When the interface software was in place, the correctness of the configuration's implementation was tested by using the standard system exerciser for INGRES. Section 3.6 discusses the implementation of the various configurations in more detail.

Of course, this approach is not without disadvantages. First of all, it is by far the one that requires the largest amount of work. Many thousands of lines of code had to be written, changed, integrated and debugged. Subtle but critical implementation problems arose and could not be finessed as would have been possible in the other approaches. These problems had to be solved. All this took time that would not be needed in the other three approaches. Analytic modeling and simulation typically can examine a broader range of parameters than an experimental approach. Each "run" of a model of this type is much cheaper, and therefore more runs can be performed. However, the greater accuracy of the results produced by the experimental approach makes this objection much less important in studies like the one described in this dissertation.

When running an experiment on real software, the performance data obtained could be relevant only to that particular software system. The following questions can be asked. Are the conclusions one may draw from the results valid for all types of database systems? Are the conclusions valid for at least all relational database systems? Are the conclusions valid for all databases and query streams or only those used in the experiment? Are they valid over the whole spectrum of hardware configurations, capacities, and over that of operating system organizations and implementations?

We believe, but cannot demonstrate, that the results obtained for some of the configurations are valid for all database systems. These are the two configurations, described below and called "access methods" and "smart disk", where the least amount of functionality was migrated to the backend. For relational database systems, we also believe that our results have a somewhat wider generality. Only one configuration, called "inner loop" below, was clearly applicable only to INGRES. Individual database systems use different methods for implementing the various database system functions (e.g., join methods, type of locking). However, the functions performed by INGRES must have some counterpart in any relational database system. Hence, the actual results from measuring INGRES are not directly applicable to all relational database systems, but the indications from INGRES do have some implications for all relational database systems. As mentioned below in section 3.8, there were two distinct databases and query streams used in the experiment. Hence, the results should not reflect the idiosyncrasies of a particular database. To enhance the credibility of the results, all the configurations were tuned. Usually the tuning was quite simple, but for two of the configurations it involved substantial changes and additional buffering, as described in more detail in sections 3.5 and 5.1.3.

However, even with these substantial drawbacks, we believe that the experimental method provides the most reliable information about how a real database system would perform in the ways we have experimented with when functionally partitioned between two computers.

3.3. The Basis for the Experiment

The University of California at Berkeley has many DEC VAX-11 series computers running the Berkeley version of the UNIX Operating System (4.1a BSD) (see Figure 3.1) [Ritc78] [Joy81b]. In addition, the INGRES database system [Ston76], that was developed at Berkeley, is available for experimentation. The Computer Science Division of the University is also currently extending the Berkeley UNIX Operating System to add networking facilities [Joy81a]. This is being built using the 3 Mhz version of the Ethernet called Research Ethernet [Metc76]. The major network and transport protocol being implemented is TCP/IP [Post80a] [Post80b]. Due to the availability of the VAX computers, of the Berkeley UNIX operating system, of the INGRES database system (version 7), and of local expertise in these two software systems, the choice of the hardware and software basis for the experiment was quite easy.

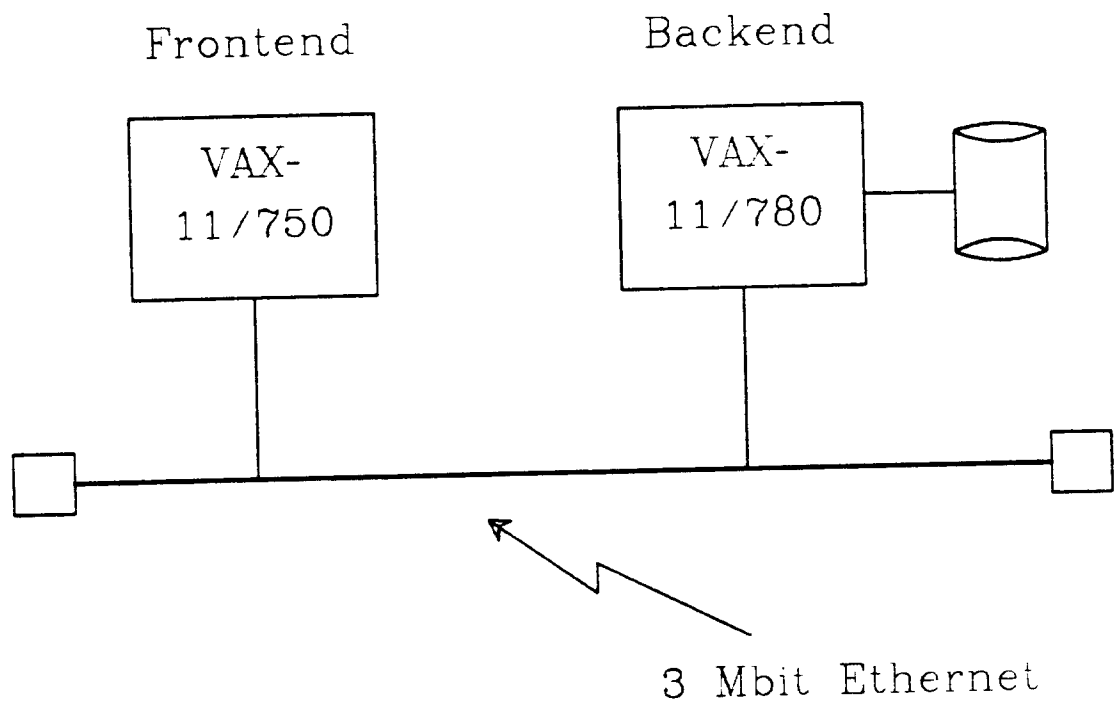


Figure 3.1: Laboratory

Two computers were used in the study, Medea and Ingvax. Medea is a VAX-11/750 and Ingvax is a VAX-11/780. Medea always served as the frontend and Ingvax as the backend.

3.4. A General Description of INGRES

Except for the utilities, Version 7 of INGRES running on the VAX-11 series under the UNIX operating system uses four processes when running with the standard terminal monitor (an ad hoc query processor). First, there is an initialization process that sets up connections for and establishes the next two processes. The bulk of the system's execution involves the process for the ad hoc query processor (*monitor*), and the process for the database system (*vaxingres*). (Both the initialization process and the monitor process may be replaced by an application program that directly interfaces to the database process *vaxingres*.) The monitor and *vaxingres* processes are connected by a pair of UNIX pipes. A pipe is a data stream between two processes on the same machine. Data written into a pipe by one process can be read sequentially by the other process. Queries coded in ASCII flow across this interface using one of the pipes. The queries are processed by the *vaxingres* process. Results are returned using the second pipe. If a sort is needed, it is invoked as a process by *vaxingres*. The sort process, *ksort*, communicates with the *vaxingres* process by reading and writing specific files in the file system. During the implementation phase of this research, the *ksort* process was merged into the *vaxingres* process to save process startup overhead.

One way to understand how a query is processed in INGRES is to follow a query through its execution. This discussion is quite brief and many details have been omitted. First, the query coded in the INGRES query language QUEL is entered into the *user interface* located in the *monitor* process (recall that the monitor process can be replaced by an application program). The user interface deals with the terminal and does some pre-processing on the query (e.g., macro expansion). The query is then passed to the *query parser* located, as are all other functions, in the main database process *vaxingres*. Here the query is parsed and checked for semantic and syntactic correctness. Protection and view processing are also done at this time. Next, the query is sent to the *query decomposition and planning* module (decomposition) who breaks the query into a sequence of subqueries that involve only one variable (i.e., one relation). Each subquery is passed to the *inner loop* (also called the One Variable Query Processor (OVQP)) for execution. The *access methods* support the inner loop by accessing tuples (records) in the database. The access methods use the UNIX file system to access the data on the disk. Results are returned to the user interface by the inner loop as they become available, or the results are written into a (possibly new) relation.

3.5. The Configurations of INGRES

As stated earlier, each subdivision of functions between the frontend and backend computers is called a *configuration*. Although a larger number of possible software configurations exist, this study focuses on the six configurations discussed below. These configurations were chosen on the basis of three criteria. The first was that almost all tightly coupled processing be done in one machine. The reason for this requirement was to make the network traffic manageable as well as to reduce the synchronization overhead needed between the two machines. Second, it must be reasonably simple to implement each configuration starting from the basic INGRES system. The final criterion was the existence of an example of the configuration either in the real world or in the literature in the form of a proposal. The configurations will be described in order of increasing responsibilities for the backend machine.

With each configuration, there is a figure that graphically represents the software division. On the top part of each of the figures, the various parts of INGRES, discussed in the previous section, are depicted. Functions that reside in the same machine are represented by having the boxes for the functions touch. Space in between boxes indicates the machine boundaries. The bottom half of each figure shows the processes used in the configuration. Solid lines connecting processes are UNIX pipes, and dashed line are TCP/IP reliable byte stream connections. The bottom half of the diagram is for use in the next section on implementation and should not be examined in detail in this section.

The first configuration, *Ingres*, is trivial to build, since it consists of the existing, non-distributed INGRES system (see Figure 3.2). It does almost all of the processing in the frontend machine. The backend machine is a standard disk controller. This configuration was tested using only one computer. The Ingres configuration was included in the study because of its historic and commercial importance, and because it provides a yardstick with which the other configurations can be compared.

The second configuration includes an intelligent disk controller and is called *smart disk* (see Figure 3.3). Here the file system functions have been moved to the backend machine. Examples of the interface are opens and closes for files, and reads and writes for disk pages. The controller has sufficient buffer space and processing power to allow it to make intelligent decisions regarding buffering policies. This configuration is represented in the commercial world by the IBM 3380 Model 13 Disk Controller [IBM81] and by the Xerox WFS file server [Swin79].

The third configuration, called *access methods*, continues the migration of functions to the backend machine (see Figure 3.4). In it, the software that gets, replaces, finds, inserts, and deletes tuples (records) from/in the database (that is the software that is normally called the access methods) also resides in the backend machine. This configuration is similar to that proposed for a CODASYL style backend database machine by Canaday et al. [Cana74], to the Navigational Interface in [Armi81], to the ADABAS Data Base Machine marketed by Software AG [Soft81], and to the IDMS backend machine built by Cullinane [Cull78].

The fourth configuration also moves the "inner loop" of query execution to the backend machine and is called *inner loop* (see Figure 3.5). Most operations on the database take the form of queries. If the query is non-trivial, the database system must read and process more than a single record. This involves looping through records and processing the data found in them. In INGRES, this is called the one variable query processor (OVQP). It is reasonable to move OVQP entirely to the backend because of the high amount of time the database spends in its inner loop for complicated queries, and the tight dependency of the inner loop on the data from the database. Dividing the inner loop and the data would tend to generate too large an amount of network traffic. The idea of moving the inner loop computation to the backend is also used in DIRECT [DeWi78] and Intel in the iDBP [Inte82]. However, the inner loop in both DIRECT and in the iDBP is much larger than the loop migrated in this configuration. A single backend call in DIRECT and in the iDBP is done to perform a join, while in the *inner loop* configuration this operation could require many calls.

The fifth configuration is called *decomposition* (see Figure 3.6). The backend machine has now become a query execution engine. A parsed and validated database query is sent from the frontend to the backend machine. The full query is executed in the backend machine. This configuration is reasonable because the interface is particularly simple: the frontend sends over the parsed query and the backend machine responds with status information and possibly a data stream. A similar organization has been adopted by Britton-Lee [Epst80] in the IDM.

In the sixth and final configuration, called *parser*, the backend computer is a full backend database machine (see Figure 3.7). Unparsed and unvalidated queries are sent from the frontend to the backend machine. The response is some status information (e.g., an error message indicating a malformed query) and possibly a data stream. Only the user interface or the application program remain in the frontend. The Relational Interface [Armi81], that is still in the prototyping stage, is a potential commercial example of this configuration. The Data Base Computer (DBC) is a university research example of this configuration [Bane78].

3.6. Implementation

With respect to the implementation, the configurations could be divided into three classes: the *Ingres* configuration that did not require any work as it was already available; some other configurations, that needed only a moderate amount of work because the functional division fell on or near an existing process or module boundary; the rest of the configurations, where this was not the case, that required substantial amounts of software modification and construction.

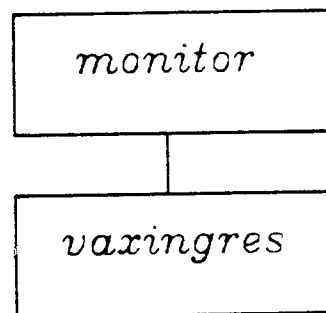
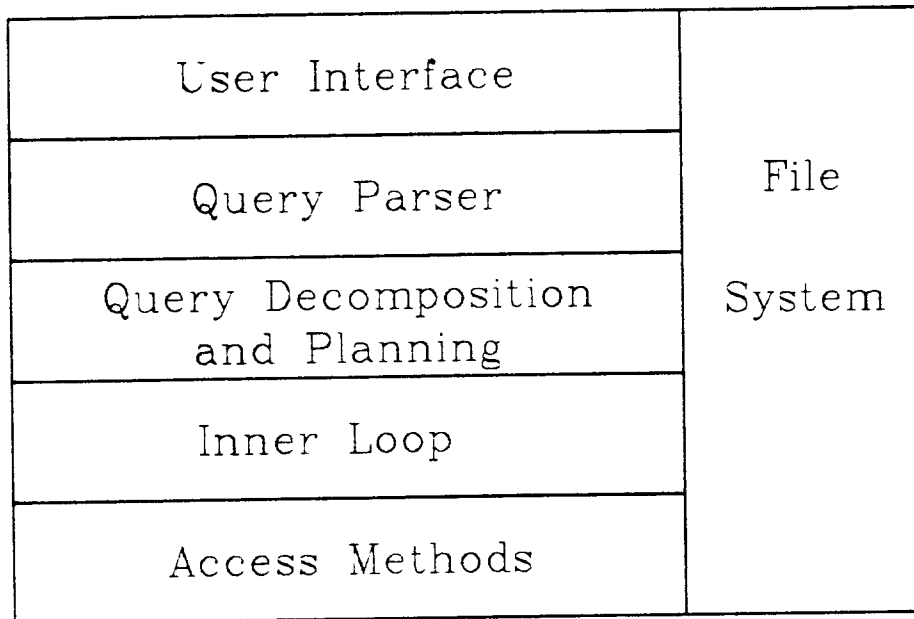
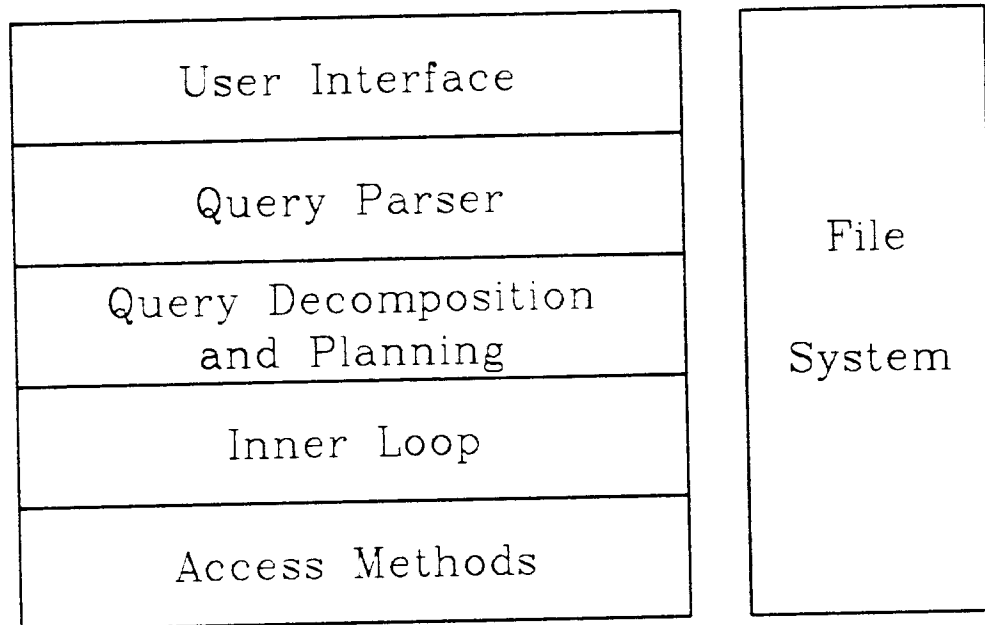


Figure 3.2: Ingres Configuration



Frontend

Backend

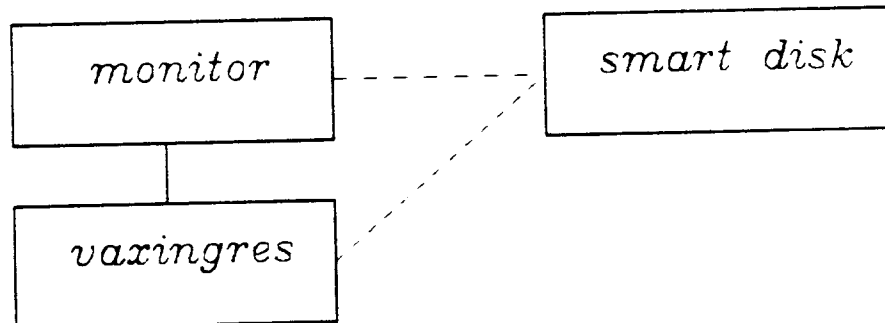
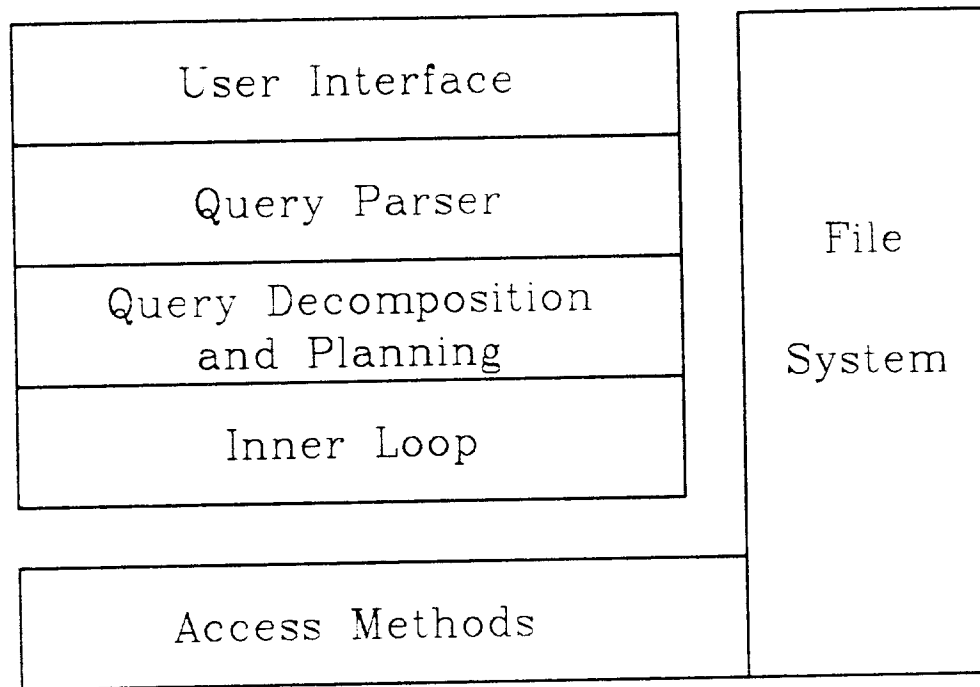


Figure 3.3: Smart Disk Configuration



Frontend

Backend

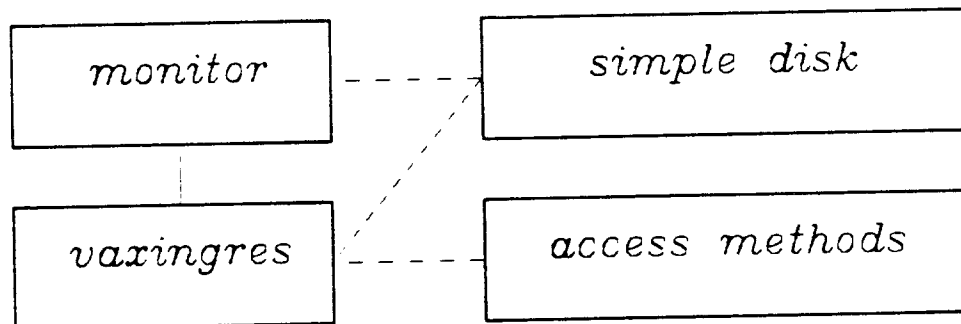
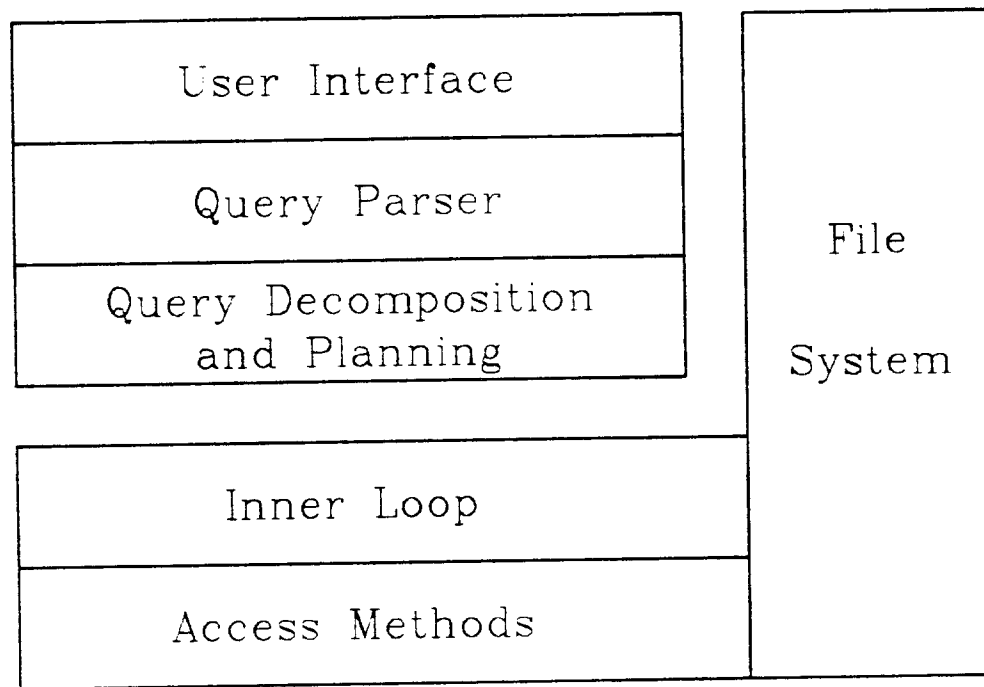


Figure 3.4: Access Methods Configuration



Frontend

Backend

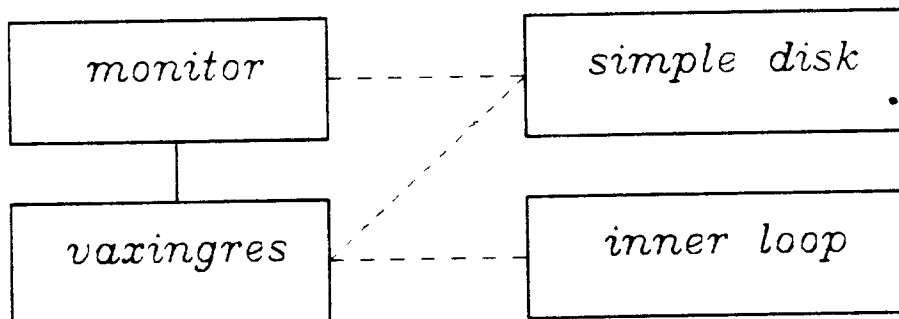
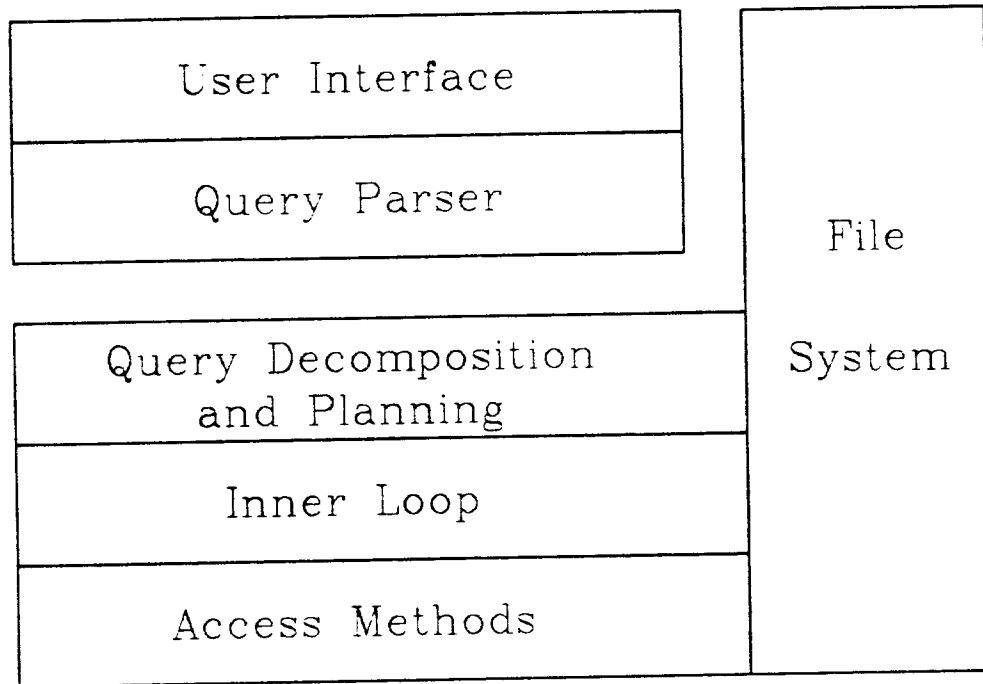


Figure 3.5: Inner Loop Configuration



Frontend

Backend

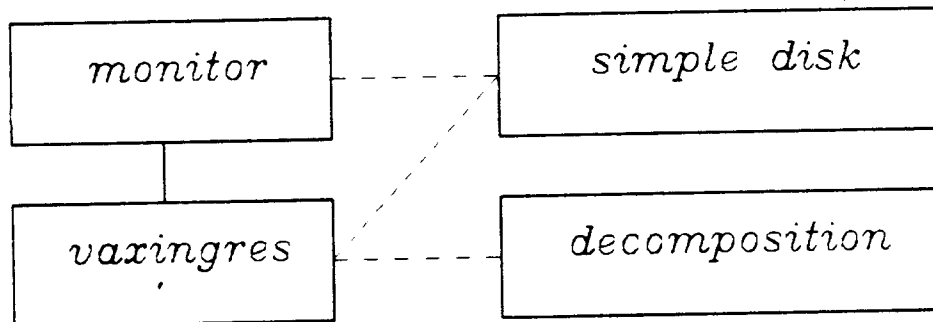


Figure 3.6: Decomposition Configuration

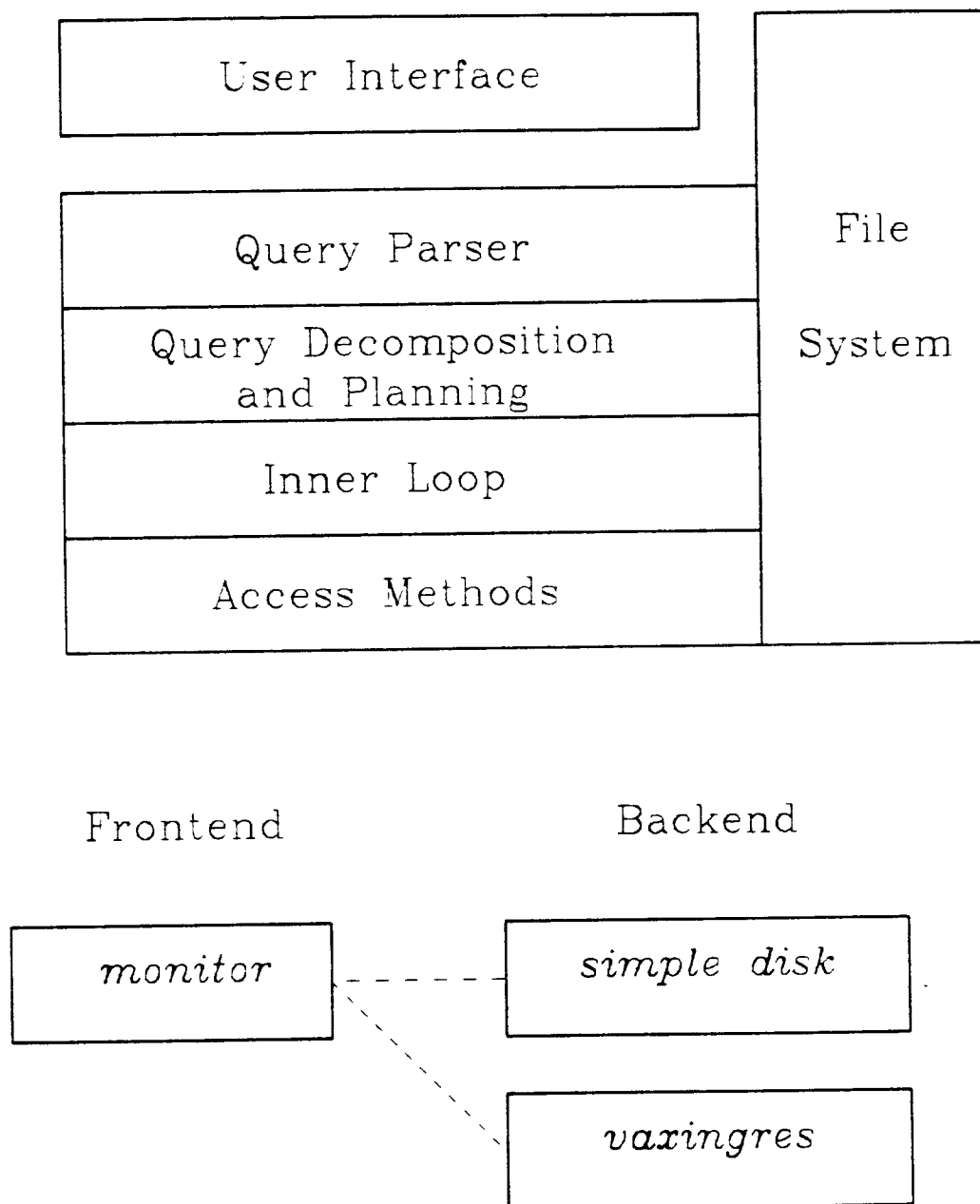


Figure 3.7: Parser Configuration

The code of INGRES is structured as a group of modules. A process is a set of modules. Part of the design of INGRES is that these modules can call each other whether or not they reside in the same process. Calls are either local within a process, or they are encoded and sent over a pipe connecting (eventually) the source process to the process that contains the module to be called. Hence, it is possible to repartition the software of INGRES into additional processes. In fact, the PDP-11 version of INGRES running under the UNIX operating system executes using one more process than the VAX-11 version.

The feature analogous to a pipe in the Berkeley UNIX operating system implementation of the TCP/IP protocols is a reliable byte stream connection. If INGRES is partitioned following module boundaries into processes on a single machine, then the pipes can be converted to connections when INGRES is run on two machines. This is, basically, what was done for the *parser* and *decomposition* configurations. Due to some peculiarities in the implementation of INGRES, the processes allocated to the frontend still have to be able to access the same file system as the backend. (This was for various functions such as storing and accessing macros and reading the text for error messages.) Hence, a simple version of the *smart disk* backend (described below) was also connected to the frontend processes. The code for the *smart disk* backend always ran in the backend.

The *inner loop* configuration also followed module boundaries. However, subtle problems concerned with file synchronization and shared global data structures made this configuration hard to build. The basic method of communication between the processes and with the *smart disk* backend was the same as for the *parser* and *decomposition* configurations.

The remaining two configurations, *smart disk* and *access methods*, did not functionally divide INGRES along module boundaries. For these configurations the following building methodology was used. First, the function or procedure calls that were to be done remotely were identified, thereby determining the functional boundary between the two machines. A "stub" was built to catch all calls from the frontend machine to functions in the backend. The stub contains a function of the same name as the function to be executed remotely. The first call to any function in the stub causes the creation of the backend process and establishes a reliable byte stream connection to it. All calls to functions in the stub are then passed to the remote process by doing a remote procedure call [Lamp81] [Nels81] [Spec82]. This means that the calls are converted to messages on the connection. Each message identifies the function to be performed and the arguments and global variables needed. The backend, upon receiving the message, determines which function is to be called, sets up its own copy of the global variables and calls the function with the received arguments. The return value of the function together with all potentially modified arguments and global variables are sent to the frontend. The frontend decodes the response and copies the arguments and global variables back to their original locations.

For the *access methods* configuration, this remote procedure call was synchronous: it blocked in the frontend waiting for the backend. The *smart disk* configuration was optimized to allow file system write calls to be non-blocking. This was done to simulate more realistically the use of non-volatile storage. However, since a write was very rarely followed by another write, this had very little effect on system performance. All other remote calls were blocking.

3.7. Buffering and Non-volatile Storage

Two additional issues in backend machine design were also addressed in the experiment: the buffering of disk sectors and the use of non-volatile storage. A relational database system keeps the data in relations. Each relation is stored on disk as a set of pages. By instrumenting a copy of INGRES and tracing the logical I/O operations to relation pages, it was possible to simulate the operations of a backend database machine from the trace data. Since the simulations were so simple, two moderate size programs were written, in C, to model the backend. Processing the trace data by simulating the least recently used (LRU) memory management policy on different memory sizes allows a study of buffer cache sizes to be performed. A simulation of several non-volatile buffer sizes was done to determine the benefits of having non-volatile storage in the backend. After a page was written to the non-volatile storage, it was transferred to disk when the

disk was idle.

3.8. Databases Used in the Experiment

Any experiment in database system performance requires that one or more databases and one or more query streams that refer to these databases be selected. In an attempt to get results that would be less tied to the particular database used in the experiment, this experiment was performed on two different databases. One was mostly synthetic, but was constructed to be representative of data and queries used in a statistical application. In particular, the application was concerned with census data and was called the Regional Accounting System (RAS) at the Lawrence Berkeley Laboratory [Keen81]. In normal operation, the RAS database is used to answer moderately complex queries. Some updating of RAS was performed, but most writes were done to temporary relations. The query stream was designed to be representative of how a true census database would be used.

The second database is called "Citator" and was obtained from the Commercial Clearing House company. Citator consists of the data used to typeset a book containing cross references between legal citations and appeals. It is updated semi-annually by appending new data. The main query for it is the one that causes a new book to be typeset. For benchmarking, only a section of the book was extracted from the database.

3.9. Network Benchmarks

During the design of the experiment, it was anticipated that for some configurations the results would be dominated by network performance. Hence, a series of experiments were designed and carried out to measure the CPU overhead and the delay associated with the network. It was hoped that a simple empirical model of network performance could be constructed. However, these tests revealed that message delay was not a nice linear function of a few selected variables (number of logical messages, number of packets, and number of bytes). Each type of remote procedure call had a characteristic message pattern: a certain number of bytes would be sent and some number of bytes would be returned. The network was benchmarked for each logical message size combination that was used in some remote procedure call used in any configuration. This was done by writing a program that first created a copy of itself on a remote machine with a TCP/IP connection between them. These two processes would then exchange a large number of messages of the desired size. Very little processing was performed by the processes in either machine. In such an environment, the average time to exchange one message could be computed. Both computers used in this experiment were dedicated and were the same machines used in the database system measurement experiments.

3.10. Criteria for Comparison of the Configurations

Part of the goal of this thesis is to keep the experiment and the measurements as general as possible. One consequence of this goal is that there are no fixed criteria that we anticipated using to compare the configurations. We also chose to run the query streams without "think time" to stress the database system as much as possible. This means that the query streams do not simulate human users, since there is a lack of think time. While the measurements taken can be used to calculate throughput, they lack the detail to compute the responsiveness to individual queries.

3.11. Measurements and Instrumentation

In a study like the one described in this dissertation, it is normal to measure the use of the following resources: elapsed time, central processor time, main memory utilization, disk I/O's, number of messages, and total number of message bytes transmitted. If at all possible, each of these statistics should be broken down into the contributions of the individual modules that consumed the resource. Since the programs were not memory limited, it was not necessary to instrument main memory utilization. This study used internal counters and statistics kept by the operating system on a per process basis as an approximation to the above ideal. To get better measurements, the hardware and/or the operating system would have to be changed, and such an

endeavor was felt to be beyond the scope of this thesis.

When getting measurements, it is desirable to obtain the same or related data in as many ways as possible, so that they can be checked against each other to insure their correctness. This experiment used three methods to obtain performance data.

While running a database system configuration, just before each process exits, it does a system call to the operating system to request a collection of performance data. This data is obtained by the "vtimes" system call, and will be called *vtimes* data in the rest of this thesis. The *vtimes* data obtained included user and system CPU time, average use of program and data space in memory, disk I/O reads and writes charged to the process, and the number of page faults and swaps of the process. This data is sent back, eventually, to the terminal monitor process. The data is then printed on a per process basis.

Each process in a configuration also keeps its own internal statistics. The number of logical reads and writes from or to relation pages was recorded. Also kept were the number of logical messages sent, an approximation of their size distribution, and the total number of bytes sent over the network.

Finally, a program was constructed to measure the total amounts of resources consumed by the system during a test. This program, *endstat*, also relied on the operating system as the collector of measurement data. It recorded total computer-wide user and system CPU time, the number of disk transfers, the number of input and output packets, and the number of input and output network errors, and packet collisions.

CHAPTER 4

Experimental Results

4.1. Network Measurements

The network used in the experiment described in this dissertation was the 3 megabit per second version of the Ethernet. At the time of this experiment, only two families of protocols had been implemented: those for shipping unreliable messages and those involving reliable byte streams. Only the reliable byte stream family of protocols (TCP/IP) was used in this experiment.

As described in section 3.6, in most cases the frontend and backend machines communicate via remote procedure calls. The arguments and the necessary global variables, preceded by a small header, were packaged in a message and written onto the stream to the backend. When this message arrived, it was unpackaged and the call was executed. The results of the procedure call plus any changed globals were then packaged into a response message and sent back to the frontend. The frontend would then unpackage the results and globals. Return would then be done to the originating procedure in the frontend.

Therefore, the interaction between the frontend and backend usually consisted of a message sent and a reply returned. The length of a message depended on the type of procedure call. By recording the distribution of message sizes, it was possible to determine the counts of the various types of exchanges (e.g., to distinguish 'open' calls from 'get' calls).

It was hoped that a simple network model could be constructed that would require few parameters and yet would be a good predictor of transmission time including the overhead. However, preliminary test results could not be made to fit well into any simple model. Hence, a separate test was run for each size combination of message (send/reply) pairs exchanged between the frontend and backend. For example, the *smart disk* configuration did a lot of page reads. The frontend would send a 40 byte message requesting that a page be read; the backend would read the page and send a 1040 byte response; therefore, to measure its performance in this type of exchange, the network was benchmarked for a 40/1040 message call/response. All the message size combinations were determined and clustered. Fifteen benchmarks were then run to get network performance data.

Due to the way connections were established, sometimes this exchange was performed on a single connection. Sometimes, however, a separate reply connection was established to minimize the changes required when converting from simplex pipes to duplex connections. The existence of a separate connection made a higher packet traffic likely, since the packet acknowledgements could not be piggy-backed on the replies. Also, a second message was sometimes sent before the response to the first had been received.

Except for a test (not shown in the table below) that exchanged 1 message, all tests made 5000 simulated calls to the backend. During measurement data collection, the only significant work that was done in either the frontend or backend was to exchange the messages. In the results presented in Table 4.1, separate reply (sep reply) entries indicate that two connections were used. The "split F/B" column means that the data was written using two writes instead of one in the frontend (F) or backend (B), respectively. A blank entry in that column means that the data was not split. The packets sent and received by the frontend include those packets necessary to start the test. Exchanging only one message caused 14 packets to be sent and 16 packets to be received in 10.9 seconds. The "adjusted time" in the table is the time it took each test to run less the single message time of 10.9 seconds. This is to eliminate process activation and communication establishment time from the total time required to do the simulated calls. Finally, the values in the exchanges per second column are computed by dividing 5000 (the number of calls) by the adjusted time.

All tests were run twice, and the results used in the rest of this thesis are the average of the two measurements. Two runs seemed sufficient since the test results were all within 4% of each

other. This indicates a high degree of repeatability, and is consistent with the 3% variability found in the database benchmarks (see the next section).

message sizes	sep reply	split F/B	frontend packets		adjusted time	exch per sec
			send	rec		
40/40			5143	5398	1:26.8	57.6
			5071	5406	1:24.3	59.3
1040/40			10128	10154	2:08.8	38.8
			10164	10347	2:14.5	37.2
40/1040			10333	10223	2:24.1	34.7
			10142	10382	2:23.6	34.8
212/160			5153	5272	1:44.7	47.8
			5185	5304	1:46.6	46.9
160/212			5124	5270	1:42.9	48.6
			5208	5268	1:45.9	47.2
128/128			5222	5305	1:42.7	48.7
			5131	5229	1:38.8	50.6
128/128	x		6013	6053	1:40.7	49.7
	x		6118	6060	1:43.9	48.1
128/256	x	B	11216	6855	2:05.7	39.8
	x	B	11196	6831	1:58.2	42.3
256/128	x	F	6770	11222	2:01.0	41.3
	x	F	6786	11241	1:59.3	41.9
2400/1032			20466	20489	4:22.7	19.0
			20238	20486	4:14.5	19.8
376/1064			10210	10325	2:46.2	30.0
			10231	10393	2:48.9	29.6
1400/200			10304	10427	2:35.5	32.2
			10243	10425	2:38.5	31.5
800/800			10216	10124	3:19.7	25.0
			10556	10185	3:21.1	24.9
28/40			5080	5395	1:27.0	57.5
			5163	5301	1:24.3	59.3
28/1064			10151	10313	2:23.3	34.9
			10158	10362	2:26.8	34.1

Table 4.1: Network Performance Data

4.2. Database System Measurements

In section 3.8, the two databases used to collect database system measurements were described. Each configuration was measured with each of the two databases separately as well as with both of them combined together. For the separate database measurements, only one simulated user performed the queries without "think time" between the completion on one query and the start of the next query. When measuring both databases together, there were two simulated users, and again no think time. Since the databases were disjoint, there was no conflict for user data between the two query streams. Hence, there was no need for concurrency control (at least not for user data).

Using two databases made the results less dependent on the actual data and queries used. Running both databases and query streams together gave some indication of the throughput increase possible with multiple machines.

The measurements reported in the thesis are for one execution of the benchmarks. Ideally, several executions should have been done to compute confidence intervals for the results. Due to the length of the benchmarks and the need for dedicated machine time, each configuration was measured only once. However, prior to the final measurement sessions, several debugging runs were performed, in particular for the *Ingres* configuration. The results from these runs indicate that there was at most a 3% variability in the major measurements: elapsed time, CPU utilization, and number of physical disk I/O operations.

This section has four subsections: CPU utilization measurements, disk I/O measurements, network overhead results, and summary of the results. Each subsection presents the results obtained with each database and with both databases running together. As mentioned in section 3.11, the measurements were collected by three instruments: the UNIX *vtimes* system call, the internal counters in the database system software, and the *endstat* program that recorded machine-wide usage.

The first column in each table is named "Config", and it contains the configuration of INGRES as described in section 3.5. Some abbreviations are used. For the RAS database, the *Ingres* configuration was benchmarked twice: once on a VAX-11/780 (*Ingvax*) and once on a VAX-11/750 (*Medea*). The results are denoted in the "Config" column by *Ing(I)* and *Ing(M)* respectively. For each configuration there is a line, where the "Config" name appears, which contains test wide results (e.g., "elapsed time"). On some tables, this line is blank except for the configuration name. All configurations, except *Ingres*, have separate lines for the frontend ("front") and backend ("back") data for all tables except for the summary tables.

Tables for the combined databases were run by executing the Citator database query stream to completion and stopping the RAS test at that time. Thus, the RAS processes did not finish normally. Because of this, only *endstat* results are presented for these runs.

4.2.1. CPU Utilization Measurements

Tables 4.2, 4.3 and 4.4 present CPU utilization data. This data was acquired by both the *vtimes* and *endstats* methods. The "Vtimes CPU" column shows the total time charged by UNIX to all the processes in the configuration. "Endstat CPU Time" reports the machine-wide use of CPU time during the test. This time is broken down into user, system and idle time. The VAX-11/750 and VAX-11/780 do not have the same performance characteristics. The *endstat* columns present the raw measurements unadjusted for differences in the processors. To get a balanced view of the CPU time expended, it is necessary to adjust the time of one machine to the other. Hence, the VAX-11/750 CPU time is adjusted to VAX-11/780 time. The *endstat* CPU times consumed when running the *Ing(I)* and *Ing(M)* benchmarks were used to determine the multiplier, that turned out to be .629. The "Total 780" times were computed by finding the total CPU time and multiplying by .629 if this portion of the test was run on a VAX-11/750. The two values of "Total 780" CPU times obtained from *vtimes* and *endstat* are given in the tables.

One problem encountered in the benchmarking experiment was an apparent bug in the 4.1a Berkeley UNIX software. A couple of the twenty-three sub-tests with the RAS database had

more characters than normal in the query inputs. The apparent bug in the operating system made the remote shell, that was sending this query data, to loop. This extra CPU time showed up principally as system time in the backend. Hence, only the user time was used to adjust the values of "net time" in section 4.2.3.

The only measurement results that need to be explained, once the above bug is discounted, are those for user CPU times measured by endstat. Here we see a nice progression of CPU cycles used from the frontend to the backend as more functionality is added to the backend (see Figure 4.1). However, the endstat user CPU time for *smart disk* (2339 seconds) is greater than the time for *Ingres* (Ing(M) used 2141 seconds). There are two reasons why this should not be very alarming. First, there is extra copying of data between buffers in the *smart disk* configuration. This would show up as increased user CPU time. Second, there is the 3% variability from run to run mentioned in the previous subsection. The combination of these two factors tends to make these measurements appear to be within configuration construction idiosyncrasies and experimental error.

The combined benchmark in Table 4.4 can be used to estimate the effects of parallelism. The configuration that had the best balance, given the relative processing powers of the two machines and network overhead, was *inner loop*. This can be seen because the idle times measured in the frontend and the backend were about the same. Of course, the balance is quite dependent on the relative processing powers of the two machines.

Config	Vtimes CPU	Endstat CPU Time			Total 780	
		user	system	idle	Vtimes	Endstat
Ing(I)	1508	1340	351	906	1508	1691
Ing(M)	2534	2141	546	769	1508	1691
Parser					1651	2775
front	935	953	333	2629	556	815
back	1095	1069	891	1967	1095	1960
Decomp					1673	2986
front	1202	1160	414	2599	715	990
back	958	980	1016	2189	958	1996
I Loop					2421	3969
front	1762	1387	845	2943	1048	1404
back	1373	1097	1468	2621	1373	2565
Access					3579	5176
front	3424	2103	1679	3787	2037	2379
back	1542	595	2402	4585	1542	2997
S Disk					2069	3570
front	2776	2339	813	1747	1652	1983
back	417	286	1301	3324	417	1587

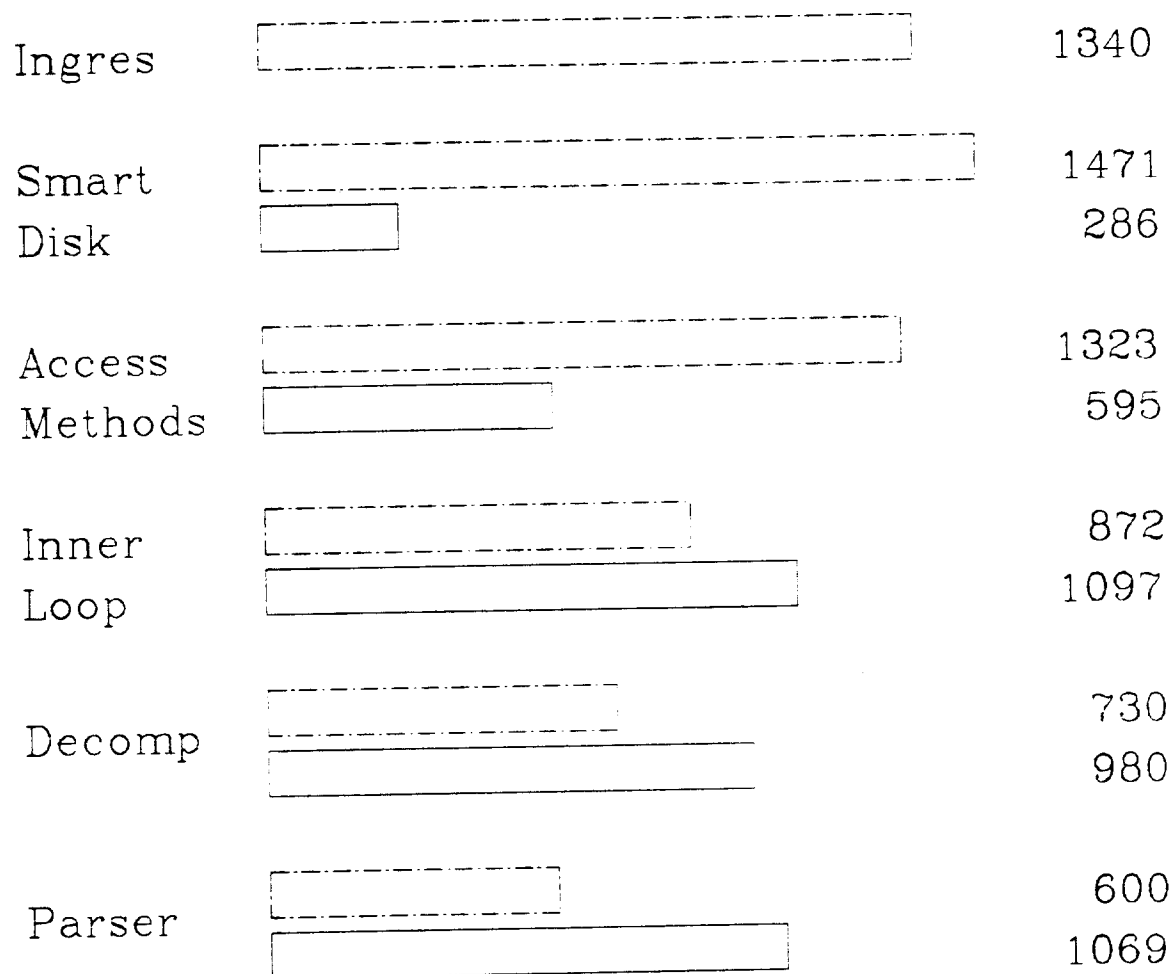
Table 4.2: RAS CPU Performance Data

Config	Vtimes CPU	user	Endstat CPU Time		Total 780	
			system	idle	Vtimes	Endstat
Ing(I)	512	313	200	554	512	513
Parser					502	517
front	8	9	6	1115	6	9
back	496	311	197	635	496	508
Decomp					508	528
front	11	12	11	1022	8	14
back	500	315	199	544	500	514
I Loop					1141	1201
front	8	234	356	1384	326	371
back	815	371	459	1157	815	830
Access					1671	1691
front	1193	450	753	2173	750	757
back	921	179	755	2457	921	934
S Disk					1026	1043
front	988	617	379	1022	621	626
back	405	37	380	1614	405	417

Table 4.3: Citator CPU Performance Data

Config	Endstat CPU Time			Total 780 Endstat
	user	system	idle	
Ing(l)	837	394	254	1231
Parser				1174
front	301	79	963	243
back	592	339	431	931
Decomp				1209
front	374	106	946	302
back	559	348	539	907
I Loop				2579
front	714	794	1458	949
back	805	825	1357	1630
Access				3678
front	1303	1691	1453	1994
back	324	1360	2784	1684
S Disk				2194
front	1633	783	410	1520
back	64	610	2171	674

Table 4.4: Combined CPU Performance Data



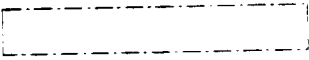
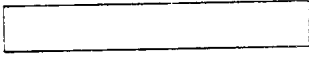
 Frontend cpu seconds
 Backend cpu seconds

Figure 4.1: RAS User CPU Measurements

4.2.2. Disk I/O Measurements

Tables 4.5, 4.6 and 4.7 present Disk I/O data. This data was acquired by all three methods: endstat, internal counters in the database software, and vtimes. The endstat disk I/O columns reflect the total number of disk I/O operations that occurred for any reason during the execution of each run. The total number of I/O's ("no."), the number of I/O's per second ("#/sec") and the estimated time the disk was busy ("% busy") are all given. The busy time was estimated by taking the average disk I/O to be 30 milliseconds. The value of "% busy" is then just three times the value in the "#/sec" column. Relation reads and writes were counted by the database software. A relation read or write is made when the database system cannot find a page it needs in its own internal buffers or has a page to be written to disk, and must request the file system to do the operation. These counts are called "Relation read" and "Relation write" in the tables. Note that these requests may not generate actual I/O operations: the page to be read may be already in the file system buffer pool. Vtimes also records all the reads and writes that UNIX charges to any process in the database software. These counts are reported in the "Vtimes reads" and "Vtimes writes" columns. Note that these three methods of measuring disk I/O activity are not equivalent. A disk I/O operation may be caused by some other reason besides those that are directly charged to database processes. A relation read or write may not cause a physical I/O because of file system buffering. Finally, other system calls besides read or write can cause disk I/O operations to occur.

As can be seen in these tables and graphically in Figure 4.2, the various ways of measuring disk I/O lead to different results. One would hope that there would be a simple relationship between the measurements from the various sources. The internal counts of relation disk I/O should be somewhat greater than the vtimes counts due to file system buffering (some reads are satisfied in the file system buffer cache). Other reads and writes to files occur in INGRES besides those to relations, but these were insignificant in our tests (5-10 read/writes per test). The sum of reads and writes reported by vtimes plus the number of disk I/O operations due to paging should about equal the number of physical disk I/O's.

Consider the *Ingres* configuration running on Ingvas for RAS (Table 4.5). Here the number of relation reads was 42,376, vtimes recorded 16,108 reads, there were 1,156 page faults (recorded by vtimes but not shown in the tables), and a total of 36,957 physical disk I/O's. The *smart disk* configuration used a 10 page read buffer. This cut the number of relation reads down to 13,996 with the vtimes count down to 12,079.

In the *Ingres* configuration, the internal count (42,376) was substantially larger than the vtimes count (16,108). However, when allowance is made for file system buffering by looking at the *smart disk* configuration's 12,079 reads, there is a discrepancy. There are about 4,000 extra disk reads charged to the process that cannot be accounted for by relation disk I/O operations. While there were 10,286 relation writes, vtimes charged the process with 16,659 writes. The total reads reported by vtimes (16,108) plus the number of writes (16,659) plus the number of page faults (1,156) is 33,865. Since there were 36,957 physical disk I/O operations, there are about 3,000 disk I/O operations not accounted for by vtimes or page faults.

There should have been up to about 12,000 relation reads, 10,000 relation writes, and 1,000 page reads. This means that there should be about 23,000 physical disk I/O operations. In fact there were about 37,000. What were the other 14,000 disk I/O operations?

Some of this discrepancy is due to the difference between what is counted as a relation read/write and a read/write as reported by vtimes. By analyzing the Berkeley version of UNIX used in the experiment, it was determined that at least the following system calls could cause the write counter for vtimes to be incremented: access, acct, chdir, chmod, chown, close, creat, dup, exec, exit, fork, ioctl, kill, link, lseek, mknod, open, pause, ptrace, reboot, stat, sync, unlink, utime, vfork, vhangup, vswapon, wait and write. It is likely that other system calls (e.g., vread and vwrite) may also cause the write count to be incremented, since the analysis program only followed simple call trees in the operating system. If these system calls branched indirect through a table of functions, they would not be included in the above list. These are almost all the system calls that deal with the file system or with process management. Note that these system calls

could, but are not guaranteed to, cause disk I/O operations. The following system calls cannot increment the write count: alarm, brk, getpid, getuid, nice, profil, setpggrp, setuid, signal, sigsys, time, times, umask, vadvise, vlimit, and vtimes. These calls are mostly used to change or to inquire about the process state, or to set or clear timers.

To process the RAS queries, only 23,000 disk I/O's were needed. An additional 11,000 disk I/O operations occurred, presumably due to system calls. That leaves 4,000 disk I/O operations that cannot be accounted for that were apparently due to internal functions in the operating system kernel. Thus, there were 48 percent extra disk I/O operations due to system calls, and 12 percent extra disk I/O operations due to internal functions in the operating system kernel. Hence, 60% more disk I/O operations than strictly necessary were performed.

This problem was even worse for other configurations. For the *smart disk* configuration, no relation disk I/O operations occur in the frontend. Vtimes reported 632 reads and 742 writes, and there were 1,348 page faults. However, the number of physical disk I/O operations counted by endstat was 10,150. The other configurations have discrepancies between 6,000 and 7,000 physical disk I/O operations. All of these discrepancies are to be attributed to the operating system kernel.

The observation is that running under an operating system is expensive. Many additional disk I/O operations occur than are needed.

Config	no.	Endstat Disk I/O		Relation		Vtimes	
		#/sec	% busy	read	write	read	write
Ing(I)	36957	14.8	44	42376	10286	16108	16659
Ing(M)	38152	11.0	33	49604	10288	16572	16553
Parser							
front	8381	2.1	6	0	0	414	729
back	41646	10.6	32	63533	10288	15875	15077
Decomp							
front	9129	2.2	7	0	0	644	759
back	41917	10.0	30	55277	10412	15983	15234
I Loop							
front	9098	1.8	5	0	0	621	759
back	44811	8.6	26	70828	11804	17390	16700
Access							
front	10903	1.4	4	0	0	710	2111
back	36002	4.7	14	29816	9858	14252	12952
S Disk							
front	10150	2.1	6	0	0	632	742
back	36230	7.4	22	13996	11355	12079	15208

Table 4.5: RAS Disk I/O Performance Data

Config	no.	Endstat Disk I/O		Relation		Vtimes	
		#/sec	% busy	read	write	read	write
Ing(I)	28876	27.1	81	53900	8689	17526	11047
Parser							
front	288	.3	1	0	0	20	15
back	28245	24.7	74	53907	8689	16650	11025
Decomp							
front	498	.5	2	0	0	26	18
back	28185	26.6	80	53656	8689	16597	11032
I Loop							
front	400	.2	1	0	0	28	16
back	33605	16.9	51	60350	9507	21233	11851
Access							
front	455	.1	0	0	0	29	28
back	28088	8.3	25	29076	8764	16488	11103
S Disk							
front	448	.2	1	0	0	31	16
back	20965	10.3	31	10772	8841	9564	11036

Table 4.6: Citator Disk I/O Performance Data

Config	Endstat Disk I/O		
	no.	#/sec	% busy
Ing(I)	45451	30.6	92
Parser			
front	777	.6	2
back	43142	31.7	95
Decomp			
front	891	.6	2
back	42823	29.6	89
I Loop			
front	1064	.4	1
back	50939	17.1	51
Access			
front	1575	.4	1
back	42294	9.5	29
S Disk			
front	1147	.4	1
back	33532	11.8	35

Table 4.7: Combined Disk I/O Performance Data

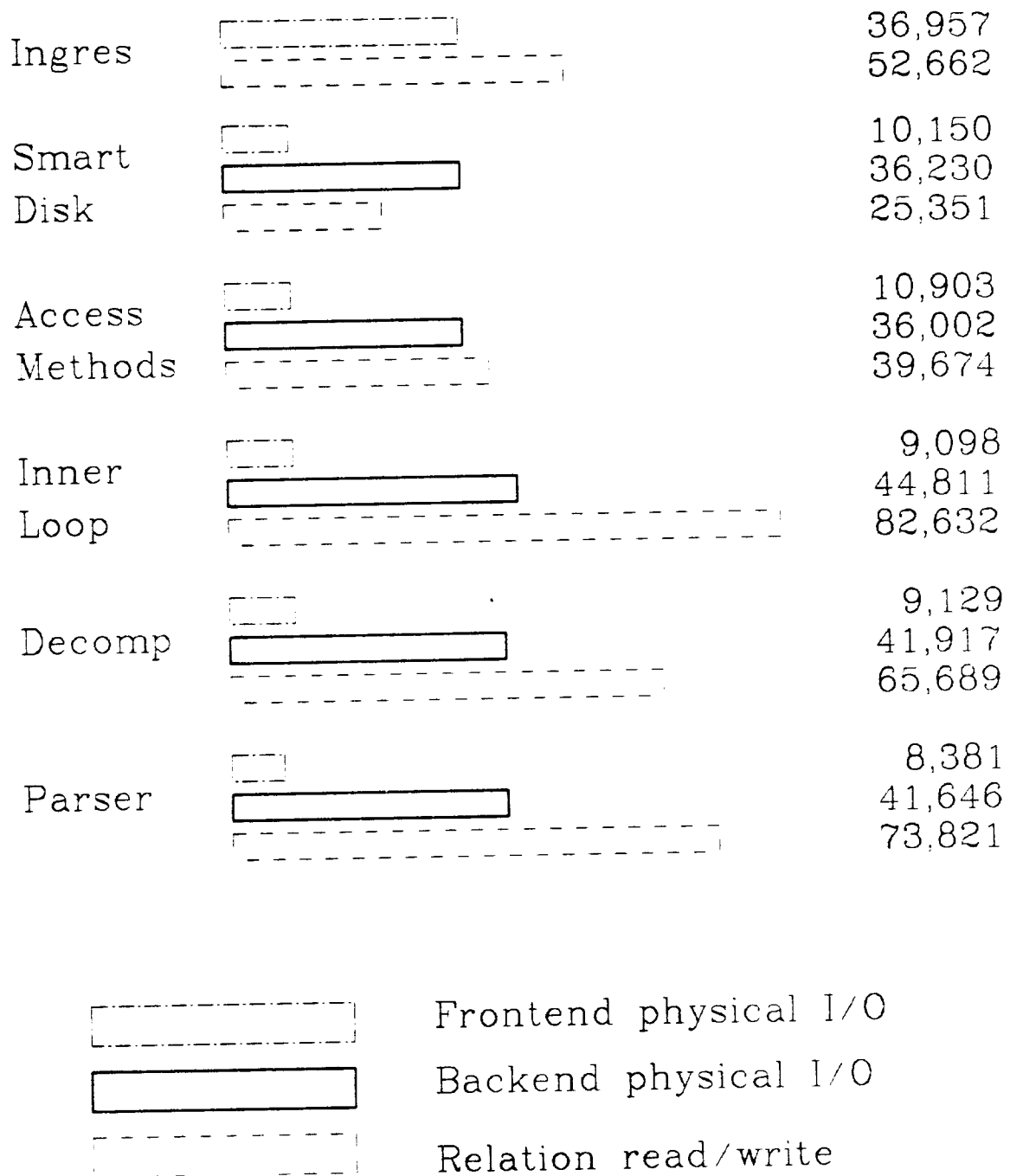


Figure 4.2: RAS Disk I/O Measurements

4.2.3. Network Overhead Results

Tables 4.8, 4.9 and 4.10 present the network overhead results. The database system has internal counters to record the number of logical messages sent ("msgs") and the total number of megabytes of message data ("megabytes"). Endstat reports the total number of output packets sent by a machine ("output pkts"). From this number and the wall clock elapsed time, the number of output packets per second ("opkts/sec") sent by a machine can be computed. The percentage of time that data is present on the Ethernet cable is computed by knowing the number of megabytes, the number of packets (each packet has a 80 byte header), and the duration of the run. This value is reported in the percent wire busy column ("% wire busy"). Finally, using the distribution of message sizes recorded internally by the processes, the count of each type of call (e.g., get a page, replace a tuple) was computed. Using the network benchmark data from Table 4.1, these counts were used to compute the total amount of time spent in purely network overhead for the test. The sum is reported in the "net time" column. Since these times were computed and not measured, additional care should be used in their interpretation.

The results reported in the tables of this subsection seem quite reasonable. Only two points are worth mentioning. First, the amount of time that the Ethernet was busy ("% wire busy") was quite low. The *access methods* was the configuration that used the network the most, and it only kept the network busy about 7 percent of the time. Second, to get the *access methods* configuration to perform reasonably well, it was necessary to tune this configuration by implementing page level buffering for "get" calls. This meant that part of the code for the access methods was duplicated in the frontend from the backend. The tuning cut the number of messages by about a factor of 5.7 and the number of bytes transmitted by a factor of 9.

Config	msgs	mega bytes	output pkts	opkts / sec	% wire busy	net time
Ing(I)	0	.00	0	.0	.0	0:00
Ing(M)	0	.00	0	.0	.0	0:00
Parser	1968	.32			.2	0:18
front			14485	3.7		
back			8370	2.1		
Decomp	7774	2.58			.5	1:33
front			22254	5.3		
back			14010	3.3		
I Loop	74936	12.54			1.7	12:00
front			71429	13.8		
back			80656	15.5		
Access	202271	142.74			7.3	53:11
front			215052	28.4		
back			203841	26.9		
S Disk	71788	26.15			2.5	14:16
front			78101	15.9		
back			68184	13.9		

Table 4.8: RAS Network Performance Data

Config	msgs	mega bytes	output pkts	opkts / sec	% wire busy	net time
Ing(l)	0	.00	0	.0	.0	0:00
Parser	33	.01			.0	0:01
front			456	.4		
back			318	.3		
Decomp	132	.04			.0	0:01
front			597	.6		
back			433	.4		
I Loop	58052	11.59			2.8	10:41
front			46640	23.6		
back			54585	27.5		
Access	133433	70.39			7.1	32:44
front			116686	34.6		
back			116664	34.4		
S Disk	65084	21.64			2.9	12:53
front			56050	27.8		
back			56088	27.6		

Table 4.9: Citator Network Performance Data

Config	output pkts	opkt / sec
Ing(l)	0	.0
Parser		
front	3666	2.7
back	2832	2.1
Decomp		
front	6749	4.7
back	5407	3.7
I Loop		
front	89037	30.0
back	94152	31.5
Access		
front	202213	45.5
back	198059	44.3
S Disk		
front	88557	31.3
back	85873	30.2

Table 4.10: Combined Network Performance Data

4.2.4. Summary Time Data

Tables 4.11, 4.12 and 4.13 present summary time data. The "Wall clock" column reports the time it took each test to run from beginning to end. The startup time for a null query stream was also computed, and subtracted the proper number of times (RAS did more than one startup) from the raw elapsed time to obtain the pure elapsed time ("wall clock less null"). The "adjusted wall clock less null" time is the "wall clock less null" time adjusted for if this test was run the test on two VAX-11/780's instead of one VAX-11/750 and one VAX-11/780. This column was computed by subtracting 0.271 (i.e. $1.00 - .629$) times the user time reported by endstat for the frontend from the "wall clock less null" time. The "net time" column is the same as in the corresponding network overhead table. Finally, the elapsed time this test would have taken if run on two VAX-11/780's, with no startup time and no network overhead, was estimated and is reported in the "adjusted wall clock" column (which does not appear in Table 4.13). This column was computed by subtracting "net time" from the "adjusted wall clock less null" time.

If there were no additional resource consumption introduced in constructing the configurations, the values in the "adjusted wall clock" column should be fairly independent of the configuration. In fact, in Table 4.11 this is nearly true for all configurations (the times range from 38:00 to 42:00) except for *inner loop*. Due to the high data synchronization costs in the *inner loop* configuration, a higher value was expected here, and the conjecture was confirmed by the results. The conclusion we can draw is that the construction of the configurations did not induce any large performance problems into the database system software, and did not, therefore, unacceptably distort our comparisons.

Config	wall clock	wall clock less null	adjusted wall clock less null	net time	adjusted wall clock
Ing(I)	39:57	37:16	37:16	0:00	37:16
Ing(M)	54:38	51:11	38:18	0:00	38:18
Parser	59:34	46:55	41:11	0:18	40:53
Decomp	63:51	48:31	41:32	1:33	39:59
I Loop	80:34	65:14	56:53	12:00	44:53
Access	120:18	107:39	95:00	53:11	41:49
S Disk	81:39	68:37	54:33	14:16	40:17

Table 4.11: RAS Summary Performance Data

Config	wall clock	wall clock less null	adjusted wall clock less null	net time	adjusted wall clock
Ing(I)	17:44	17:37	17:37	0:00	17:37
Parser	18:50	18:17	18:15	0:01	18:14
Decomp	17:26	16:46	16:42	0:01	16:41
I Loop	32:54	32:14	30:50	10:41	20:09
Access	56:17	55:44	53:02	32:44	20:18
S Disk	33:38	33:04	29:21	12:53	16:28

Table 4.12: Citator Summary Performance Data

Config	wall clock
Ing(I)	24:42
Parser	22:26
Decomp	23:50
I Loop	49:30
Access	74:10
S Disk	47:09

Table 4.13: Combined Summary Performance Data

4.3. Page Buffering in the Backend

As stated in section 3.7, a copy of INGRES was instrumented to get a trace of relation page usage. One of the two studies performed on this trace data was to examine the effect of a cache for page level buffering. The trace data was used to drive a simulator written explicitly for this purpose. A least-recently-used (LRU) policy was used to manage the cache. Although LRU is not the optimum replacement policy for INGRES, its performance is within 10 to 15 percent of those policies studied in [Kapl80], and it is very simple to build a simulation program for this policy. The cache size was varied to measure the effectiveness of very large buffers. Pages updated in the cache but not on disk were written to disk at the end of each transaction.

The trace data was processed in two ways: caching all relation reads and writes, or caching all reads and writes to non-system relations (i.e. all relations except for "attribute" and "relation"). Section 4.7 below stresses the importance of handling the system relations specially.

Tables 4.14 and 4.15 present the results for all relations and for only non-system relations respectively. The "Number of Buffers" is the number of one page buffers in the cache. The "reads" and "writes" columns report the counts of those operations that are needed with this level of buffering. The line marked <actual> shows the actual numbers of read and write requests made by the database system.

What is striking about these results is that much of the file system I/O is caused by system relations. Also, a single page buffer can reduce the numbers of the reads and writes by a large factor. There is clearly some performance bug in INGRES. (Note that this performance problem is actually mitigated in the UNIX file system buffer cache, so the effect is not very large in the released version of INGRES.) The addition of buffers did not increase performance to a large extent: there is no knee in the table. The use of about 20 page size buffers seems adequate, but this must be evaluated in the context of the tradeoff between the cost of memory and the increase in performance.

Number of Buffers	Reads	Writes
<actual>	80,208	13,141
1	26,785	9,843
2	24,593	8,226
3	22,028	6,270
4	20,367	5,517
5	19,678	5,255
6	18,973	5,047
7	18,265	4,852
8	17,852	4,729
9	17,399	4,568
10	16,814	4,419
20	14,224	3,450
30	12,566	2,920
40	11,184	2,516
50	9,990	2,199
100	6,798	1,386
200	4,339	958
500	2,875	790
1000	1,815	641
10000	1,184	556

Table 4.14: Disk Page Buffering for All Relations

Number of Buffers	Reads	Writes
<actual>	45,869	6,839
1	13,660	3,506
2	12,874	3,169
3	12,262	2,895
4	11,987	2,799
5	11,767	2,730
6	11,653	2,645
7	11,413	2,550
8	11,225	2,518
9	11,166	2,466
10	11,050	2,454
20	9,926	2,017
30	8,941	1,762
40	7,637	1,502
50	6,931	1,374
100	5,577	1,073
200	3,451	762
500	2,467	675
1000	1,767	588
10000	1,121	518

Table 4.15: Disk Page Buffering Without System Relations

4.4. Non-volatile Storage

The second study performed on the trace data mentioned in the previous subsection was a simulation of non-volatile storage. The results of the simulation are presented in Tables 4.16 and 4.17. Here the backend is assumed to contain buffer space ("Total Buffers"), some of which is volatile and some is non-volatile ("Non-volatile"). Only reads and writes to non-system relations were simulated. As write requests arrived, they were cached in the non-volatile buffers. If no non-volatile buffers are available, the contents of a non-volatile buffer must be written to disk while the current request waits ("Write Wait"). The policy simulated for writing the non-volatile buffers to disk was to perform the write when the simulation of the disk showed that the disk was idle. Hence, a read arriving while a buffered write was in progress must wait if the page needed was not in the backend's cache ("Read Wait"). The interarrival of read or write requests from the trace was modeled as a Poisson process, with a mean arrival time of 25 milliseconds for Table 4.16 to model the current INGRES system, and 3 milliseconds for Table 4.17 to model a faster system (in terms of CPU speed). The "Elapsed Time" column is the total wall clock time the simulation covered. All times in the tables are in seconds.

There are several conclusions that can be drawn from the simulations. First, the conflict for the disk as measured by "Read Wait" is larger for simulations with an interarrival time of 3 milliseconds than for those with a time of 25 milliseconds. This is reasonable since in the former case, the time required by a disk write is nearly the same as the mean request interarrival time. For an interarrival time of 3 milliseconds, there is a much larger chance that a conflict for the disk will occur. Second, the "Elapsed Time" for most tests with one non-volatile buffer was about 10% faster than with no non-volatile buffers. Although additional non-volatile buffers increased performance, the change was never as dramatic as the change from 0 to 1 non-volatile buffers. Hence, the inclusion of non-volatile memory in the backend can help performance, but the amount of such memory may be kept quite small. A rule of thumb might be that the backend have one non-volatile buffer for each transaction expected to be executing concurrently.

Total Buffers	Non-volatile Buffers	Elapsed Time	Read Wait	Write Wait
10	0	1,798	0	205
10	1	1,612	4	14
10	2	1,606	11	2
10	5	1,606	12	1
20	0	1,777	0	205
20	1	1,590	4	14
20	2	1,583	10	2
20	3	1,583	10	1
20	4	1,583	10	1
20	5	1,583	10	1
50	0	1,706	0	205
50	1	1,517	2	14
50	5	1,508	6	1
100	0	1,671	0	205
100	1	1,482	1	14
100	2	1,472	4	2
100	3	1,472	4	1
100	5	1,472	4	1
100	20	1,471	4	0
100	50	1,471	4	0
1000	0	1,562	0	205
1000	500	1,359	0	0

Table 4.16: Non-volatile Page Buffering
with a Mean Interarrival Time of 25 Milliseconds

Total Buffers	Non-volatile Buffers	Elapsed Time	Read Wait	Write Wait
5	0	652	0	205
5	1	552	27	77
5	2	541	41	53
5	3	537	45	44
10	0	640	0	205
10	1	538	25	78
10	2	527	39	53
10	3	522	42	45
10	4	521	44	42
20	0	619	0	205
20	1	515	23	79
20	2	503	35	54
20	3	499	39	47
20	4	498	41	43
20	5	497	42	41
20	10	495	45	36
100	0	513	0	205
100	1	400	10	82
100	2	385	18	59
100	3	381	20	53
100	4	380	21	50
100	5	379	22	48
100	50	373	33	32
1000	500	235	21	14

Table 4.17: Non-volatile Page Buffering
with a Mean Interarrival Time of 3 Milliseconds

4.5. Granularity of Communication

Communication between a frontend and a backend machine can be expensive, in terms of both throughput and latency. If the communication medium has a relatively small bandwidth, it is very important to send as few bytes as possible over the interface. If communication costs are high, then it may cost more to send a request to a remote machine than to process it locally.

Thus, very careful planning of the granularity of communication (the message size) is necessary in a distributed processing environment. In the tests for the *access method* configuration, it quickly became very clear that the natural granularity of communication made the configuration so slow that it was impractical to benchmark. With the addition of page level buffering, a dramatic improvement occurred (a factor of about 5 in wall clock time). This improvement was directly due to the high costs of running TCP/IP and the necessity of executing some low level protocol functions in the VAX (e.g., byte swapping and checksum computation).

Not all networks have the same characteristics. The critical factor may be the number of messages, the total number of bytes exchanged, the total number of packets, or the bandwidth of the communication medium. A factor may be critical because of an unexpected phenomenon: for example, the number of messages may be the bottleneck not because of network considerations but because of the context switching overhead each message causes in one or both of the machines.

4.6. Performance Statistics under UNIX

UNIX makes two types of performance statistics available: counts of events and sampled quantities. The event counters are quite accurate, but they have two problems. First, the operating system software used in the experiment typically takes seven seconds to start a process on a remote machine. This means that statistics that were to be coordinated between the frontend and the backend had a 7 second gap. This decreased the accuracy of the elapsed time recorded for a test, and added extra items to certain counters during the process startup period. Second, counters do not always work exactly as described in the manual. As discussed above in section 4.3, the disk I/O reads and writes charged to a process could come from a multitude of sources. More accurately, the disk read/writes attributed by UNIX to a process are those physical disk I/O operations whose cause can be traced to that process, even though the counter was specified in the manual as counting reads and writes.

The only sampled quantities used in this experiment were CPU utilization variables. There are two basic problems with the sampling. First, the clock used for sampling is also used to drive internal events in the operating system. Since these events trigger activities that are never seen as running by the sampler, they are never recorded. Second, when the sampling occurs, the currently running process is recorded as being active either in system or user state depending on the processor state. If the machine is in system state, attributing this sample to the system time of the currently running process may be inaccurate. If the system state was entered because of a system call, then this is reasonable. If, however, the system state was entered via an interrupt, then charging the currently running process is questionable. If a process sends a message to a remote process, then the sending CPU time is correctly charged, but the receiving CPU time is attributed to whatever process is running at the time the packet arrives. Since this is unlikely to be the process that receives the data, charging the CPU time to this process is unreasonable.

Consider the example of an I/O bound process and a CPU bound process running in the same machine. The I/O bound process is almost never charged for the interrupt time spent for its I/O. Almost all of this time is likely to be attributed to the CPU bound process since it is almost always running when the I/O interrupt occurs. Another example is that of an I/O bound process on an otherwise idle computer. All interrupt time is now assigned to the null process or to the system itself. The CPU time statistics will tend to understate the true CPU cost of an I/O bound process, and to overstate the CPU cost of a CPU bound process.

All of the above explains why the sum of the vtimes CPU times does not always match the endstat CPU time. It also is a good example of why it is desirable to measure the same quantity in several ways so that confidence can be gained in the accuracy of the measurements.

4.7. The INGRES Process ID

When the INGRES database system needs to construct for its own use a temporary relation, it builds a unique name based on the process identifier of one of the database processes. The temporary relation name and its field names are stored in the system relations "relation" and "attribute", that are stored by hash value. Since the hash value can change from test to test, the access to the system relations is different from test to test, thereby making the tests non-repeatable and subject to error. Above, the overall estimate was made that the variability was within 3% for the entire test and for all tests.

The temporary relations mentioned above caused disk I/O anomalies. Various runs of a sub-test of the RAS database for the *Ingres* configuration gave relation read counts of 628, 655, 655, 899, 1267 and 1690. Of these, only 345 went to relations other than the relation relation and the attribute relation. The reads to the system relations accounted for 45 to 80 percent of all relation reads in this sub-test. The vast majority of these reads were due to lookup of temporary relation data. While measuring the *Ingres* configuration, we found that the percentage of reads and writes for the entire RAS benchmark that went to system relations of all file system reads and writes were 43 and 48 percent respectively.

It is surprising to note that many of these relations are internal temporary relations. They are unnamed, as far as the user is concerned, and they do not survive the transaction they are involved in. Although it is convenient to keep all the relation descriptors together in the relation and attribute relations, there is no compelling reason to do it. INGRES could have two different types of relations: normal relations cataloged in the relation and attribute relations, and special relations for temporaries. The catalog information for temporary relations could be managed separately. This could eliminate up to 40 percent of the file system calls.

CHAPTER 5

Discussion

5.1. Configuration Comparisons

Each configuration consumed different amounts of resources in the experiment. Depending on the performance metric chosen as the most important one, each configuration could be considered the best performer. No configuration was uniformly better than all other configurations based on the tests described in the previous chapters. The parameters of the communication medium and the protocol overhead, the relative processing powers of the frontend and backend, and their respective processing loads can cause one configuration to outperform all other configurations. In addition, certain design factors of a backend database machine for navigational database systems (e.g., IMS and CODASYL) make some configurations infeasible. This is because the semantic level of the interface between an application program and the database system is quite low in navigational database systems. Hence, the higher level interfaces, those found in the *inner loop*, *decomposition*, and *parser* configurations, could not be built for these types of database systems. This section examines each configuration and describes the environment in which it is expected to perform best.

It should be noted that almost all the factors referred to in this section are related to performance or cost. In practice, other factors are certainly important: market place acceptance, overload of the frontend, protection, portability, and maintainability should all be considered in building a real world database backend.

5.1.1. Ingres

This configuration was included as a reference; it is not a serious contender for a backend database system. However, the *Ingres* configuration has two advantages over all other configurations: it can be built using only one machine and uses no additional hardware components. If the query load is light, the extra cost of a more complex backend than a simple disk controller may not be justified. Even a smart disk controller may not be cost effective if the type of query load does not lend itself to effective buffering (e.g., semi-random single tuple retrieval out of a large database). *Ingres* has the advantage over the other configurations in that all decisions are made in one machine. This configuration used the fewest total CPU cycles. The reason for this is simple: the communication and the coordination needed in the other configurations are not free.

5.1.2. Smart Disk

The *smart disk* configuration did not perform as well as expected. A small amount of buffering worked quite well, but its effectiveness did not scale to larger buffers (see section 4.3). Including stable storage in the controller can be very effective, but only if the query load does a comparatively large number of updates (see section 4.4).

We feel that the following properties are necessary in a smart disk controller to be used as a database system's backend. First, the controller must possess an accurate model of the processing order of the disk pages [Ston81]. When reading a relation serially, the access to pages may not be sequential from the viewpoint of the file system or the disk controller. Some inserted tuples cause disk pages to be split and an overflow page created. These overflow pages are to be logically inserted into the body of the file when it is read serially. Databases thus have a logical order of pages that may be quite different from the physical order of sectors on a disk or even from the logical page order of a file. If the controller knows that sequential processing of a relation is needed, then better buffering than simple sequential pre-fetch can be done at the controller. The controller can either learn that sequential access is required from a hint from the frontend or detect sequential processing by itself.

Second, the controller should have sufficient stable storage to perform the delayed writing of some amount of committed data. Note that only committed data for non-temporary relations needs to be buffered in a stable manner: a system crash can destroy database internal temporary relations without any major consequences. An estimate of the amount of the data to be buffered can be obtained by measuring the effectiveness of the delayed write policy in the operating system. The size of the buffer can be determined by doing a cost/performance analysis.

Third, multiple pages should be read from disk when the cost of doing this is sufficiently low, and when the processing order of pages makes it likely that some of the additional pages will be used.

Finally, the disk controller must do "load through". As stated above, when a page is needed that resides only on disk, it is likely that the controller will choose to buffer more data than that actually requested. However, it must send the requested page to the frontend as soon as possible.

A hypothetical example can be used to describe this last point. Suppose it is the policy of the controller to buffer up to eight pages for each read, and suppose we need page 3 of this buffer. When the head comes on the cylinder where the pages to be read reside, suppose that the head is just before the beginning of page 1. If the controller does no data transfer until the disk turns so that page 0 of our buffer is under the head, we will lose almost an entire disk revolution. Suppose instead that pages 1 through 8 are read. If the data is not sent to the frontend as soon as it is available in the backend (i.e. at the time page 3 is in the controller's buffers), we can lose a good part of a revolution (the time from page 3 to page 8). In this example, the best policy is to transfer pages 1 through 8 into the controller's buffer. As page 3 is read off of the disk, it should be sent immediately to the frontend (load through). Page 0 may be added to the buffer if the head does not have to be moved before the next physical I/O operation. The load through policy consists of reading and transmitting to the frontend the data page needed as soon as possible, and to transfer into the buffer nearby pages if it is convenient.

5.1.3. Access Methods

The *access methods* configuration does not make sense for relational database systems. For navigational database systems, it is hard to imagine circumstances where this configuration performs well. As the network performance results in chapter 4 indicate, the *access method* configuration caused the heaviest load on the network. Before page buffering was added, this difference was even more extreme. (Buffering five pages in the frontend cut the number of messages by a factor of 5.7 and the number of bytes transmitted by a factor of 9). Clearly, the *access method* configuration requires a high-bandwidth, low overhead interface.

The market for this type of backend machine may be quite narrow. The backend must be a moderately powerful computer, since it has to perform fairly complex tasks. Thus, it cannot be an inexpensive machine. A high bandwidth and low overhead communications medium must be used (e.g., a bus or parallel I/O channel). When a backend is being built to support an existing database system, this configuration can efficiently run only if the access methods are on a process boundary in the original database software. For new systems being built, the process structure should satisfy the same requirement. This may very well be the case in IMS or CODASYL style database systems. The underlying reason for this structural requirement is that the database system must protect itself from the user, and the only real protection mechanism in most systems is that of process isolation. In IMS or CODASYL style database systems, the interface to an application program is at the access methods level. We speculate that the main savings for the frontend when the access methods are moved to the backend are not primarily in user CPU time, but rather in context switch overhead time. Each call to the database usually causes two full context switches: one to the database process and one back to the user process. By moving the access methods to the backend, the two full switches become two half switches: a half switch to the operating system to issue the request to the backend, and a half switch from the operating system back to the user process when the request is completed. (A full context switch is a change from one process to another process; a half context switch is either a process giving up control to the

operating system kernel without starting a different user process, or the operating system giving control to a user process.). The overhead of communications must be much less than the savings due to context switch overhead. The cost of this type of backend must be offset by saving CPU cycles in the frontend. While this is possible in IMS and CODASYL style database systems, it is unlikely to be effective in a relational database system.

5.1.4. Inner Loop

The *inner loop* configuration used more CPU time in both the frontend and backend than the *decomposition* configuration. It is, therefore, never to be preferred to *decomposition*.

This configuration resulted in the best distribution of the CPU usage between the frontend and backend. However, the amount of coordination required between the two machines made this configuration almost impractical. Although initial analysis of this configuration made it appear to be viable, testing proved that this was not the case. We feel this was because the level of the interface was too low: the backend should have been able to do joins in response to a single request from the frontend instead of usually requiring several requests.

5.1.5. Decomposition

This configuration was quite successful, as it combined a reasonably low amount of communication with high processing locality in the backend. If the backend is built to perform reasonably well, this configuration should be expected to do quite well. Unfortunately, this configuration is only applicable to relational database systems.

Both this and the *parser* configuration provide a high level interface to the backend machine: the semantic content of the messages exchanged by the two machines is large, the messages are fairly infrequent, and the context switch time consumed is small. Both configurations also have a good match between the protection boundaries of the database system, and the physical boundary between the two machines. In the *decomposition* configuration, the backend must check that the parse tree passed to it has certain properties. This is necessary to insure that the parse tree makes sense; and to provide some protection in the backend, which may be necessary if the backend is to be interfaced to multiple frontends. Most of this checking must be done in the parser anyhow, but in the *decomposition* configuration some of the checks must be duplicated in the backend.

5.1.6. Parser

Like the previous configuration, *parser* was successful, but is only applicable in relational database systems. Since it had the lowest communication costs, by far, of any of the configurations, this configuration could be used in contexts in which the communication medium is slow or the protocols expensive. Except for communication costs, it performed very much like the *decomposition* configuration. In fact, our experiments suggest that the performance difference between these two configurations is probably smaller than the effects caused by our design decisions and by the errors affecting our measurement results. A cost advantage in the backend, the network communication speed, or a bottleneck could make either configuration preferable to the other.

As noted in the previous sub-section, in this configuration the natural database protection boundary nearly coincides with the machine interface.

5.2. Experiment Versus Modeling or Simulation

As discussed in section 3.2, this thesis used the experimental approach. The main reason given there was that such an approach would give the best results since building accurate models would be too hard and running them too expensive. This section reviews that decision on the basis of the experimental results.

Some of the configurations were quite hard to build and debug. The functional divisions specified in section 3.5 do not really describe the problems of the actual implementation. The

performance of the system can be quite adversely affected by a migration of some (apparently) small functions between the frontend and backend (e.g., adding page level buffering in the frontend for the *access methods*). To get reasonable performance data, each configuration was to be tuned. This resulted in some minor functions being moved between the two machines; in some cases, a cache was added to the frontend.

Some features that are conceptually quite easy are tremendously hard to implement and debug. Examples of these features are those needed to ensure file and buffer consistency. Details of the implementation, such as internal relation numbers, must be kept consistent between the frontend and backend. All this entails additional code to be written and run, and extra messages or data to be exchanged between the frontend and backend.

All of these problems make it very hard to build a realistic analytic or simulation model.

In addition to the objections above, some of the performance results show that the configurations do not run as would be expected based on the basis of simple assumptions. About 40-50,000 relation reads were found to be necessary in the *Ingres* configuration processing the RAS benchmark. By use of 10 page buffers, the smart disk configuration was able to reduce this to about 14,000 relation reads. Since the UNIX file system buffers are far more numerous (about 50), less than 14,000 physical reads to relation pages were done in the *Ingres* configuration. Since there were also about 10,000 relation writes, the total number of relation disk I/O operations should have been about 24,000 or less. However, 37-38,000 disk I/O's were counted in the operating system kernel. The previous chapter examined the causes of disk I/O, but the point here is that this operating system overhead would not have been predicted by a simple model of database use.

The conclusion is that the proper choice was experimentation. Even very sophisticated models built without having confronted practical implementation problems beforehand would have ignored several significant aspects of an actual database system. The experience and insight gained by really building the software was quite different from that needed to plan the configurations. Even the careful measurement of a single-computer database system would not have been nearly as satisfactory as actually building the distributed software. Despite the implementation problems we encountered, we feel that the experimental approach gave us the most reliable results.

CHAPTER 6

Conclusions and Future Work

6.1. Conclusions

The prime goal of this thesis was to investigate the relative performance of several functional subdivisions of a database system. Depending on the choice of performance metric, the relative processing power of the two computers, the network overhead and other factors, there is no configuration that is uniformly better than all others. The *smart disk* configuration should be used where there is a strong concern with overloading the backend since it uses the least processing power in the backend. The *access methods* configuration could perform well for IMS or CODASYL style databases provided that communication is very inexpensive. The *decomposition* and *parser* configurations were both found to be excellent choices for relational database systems. The data collected in this study could not adequately distinguish them. Hence, each configuration, except *inner loop*, is to be preferred to all other configurations in certain cases or contexts.

Our method of investigation was based on experimentation. The results of the measurements of operating system overhead due to disk accesses would alone suffice to prove that this method of investigation is more reliable in this case than simulation or modeling.

Several times during the debugging and benchmarking of the configurations, the direct or indirect measurement of a given quantity by multiple techniques proved to be very useful. Not only did this point out errors in measurement, but it also caught operating system performance bugs and uncovered anomalies in disk I/O and CPU usage.

We would also like to caution other performance investigators who use the UNIX operating system: beware the performance statistics presented by the system! The statistics that are measured by sampling are suspect. Statistics which are obtained by the use of counters also have a problem: the description of what the counters measure are often misleading.

The INGRES system did not run queries the same way every time, since the system needs a unique identifier and it constructs one from the process identifier. This caused some trouble for someone interested in benchmarking the system for performance since tests are not repeatable. Any system that uses unique identifiers, such as process identifiers, has a similar problem that an experimental performance investigator must address.

The network protocol dominated the performance of one of the configurations and was quite significant in two other configurations. Although we did expect the protocol to have a significant impact on performance, the degree that it influenced performance in some configurations was not expected. To send a one packet message takes around 10 milliseconds (i.e. about 10 times the context switch time). It is beyond the scope of this thesis to examine in detail the causes of this phenomenon.

A second interesting feature of the network performance is that it is not a simple function of the number of messages, the number of packets, and the number of bytes sent. The network seemed to be improving its performance as messages got longer.

The use of large buffers in the backend did not seem worth the expense. Although the backend performed somewhat better with more space, the use of twenty page buffers seemed more than adequate. This does seem counter-intuitive and may only be an anomaly of INGRES.

Non-volatile storage in the backend helped transactions to commit quickly, but it would not help read-only transactions. In many workloads, read-only transactions dominate the use of the database. Also, transactions that modify large sections of the database will not be helped unless the amount of non-volatile storage is large. The conclusion is that non-volatile storage is helpful but will not dramatically change the performance of a database system unless the application is very specific. This conclusion should be re-examined for distributed database systems in which

the commit protocol needs to write small amounts of data often and reliably.

One of the most impressive results was the size of the operating system overhead. One measurement indicated that 60% more disk I/O occurred than was necessary (i.e. about 40% of disk I/O was overhead). We speculate that there is a similar large overhead in CPU cycles. We feel that these overheads were primarily due to the mismatch between the database system and operating system requirements. The conclusion is that a backend should not run under a standard operating system. The performance penalties of using a standard operating system are too large. Instead, a specially tailored runtime executive is about all that is needed. Functions that are candidates for omission from the executive are memory management, device management, processor management, and the file system. Memory management can be performed better by the database system since it can make a more intelligent use of buffer space. Device management for the disk is important because of crash recovery. Processor management by the database system can help prevent convoys. Finally, the general file system provided by typical operating systems is too rich in functionality and too expensive to run with a database system.

We also observed in INGRES that there appears to be a tradeoff between modularity and performance. By adding some semantics of the higher level functions to the low level implementation, a gain in performance can be achieved. This can be seen in disk page buffering, in adding the concept of temporary files to a file system, and in disk head scheduling.

6.2. Future Work

At the beginning of this work, we did not have sufficient data to build accurate models of a database system, or in particular of the INGRES database system. Our work can now be extended to build a model of INGRES based on the data obtained in our experiments. A more general model of relational database systems based on the underlying data model could also be constructed.

A further extension of this work would be to test additional configurations. In particular, a configuration similar to inner loop that also did join processing in a single call to the backend would be interesting.

Bibliography

- [Ampe82] *Capricorn Models 165/330 Disk Storage Drive Operation and Maintenance Manual*, 3312369-01, Revision D, Ampex Corporation, El Segundo, California, May 1982.
- [Armi81] J. P. Armisen, J. Y. Caleca, *A Commercial Back-End Data Base System*, Seventh International Conference on Very Large Data Bases (1981), 56-65.
- [Aror81] S. K. Arora, S. R. Dumpalt, K. C. Smith, *WCRC: An ANSI SPARC Machine Architecture for Data Base Management*, Proceedings of The 8th Annual Symposium on Computer Architecture, (SIGARCH Vol. 9, No. 3, May, 1981), 373-388.
- [Astr76] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. P. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, V. Watson, *System R: Relational Approach to Database Management*, ACM Transactions on Database Systems, Vol. 1, No. 2 (June 1976).
- [Babb79] E. Babb, *Implementing a Relational Database by means of Specialized Hardware*, ACM Transactions on Database Systems, Vol. 4, No. 1 (March 1979), 1-29.
- [Banc80] F. Bancilhon, M. Scholl, *Design of a Backend Processor for a Data Base Machine*, Proceedings of the ACM SIGMOD 1980 International Conference on Management of Data, Santa Monica, California, May 1980, 93-93g.
- [Bane78] J. Banerjee, D. Hsiao, *Performance Study of a Database Machine in Supporting Relational Databases*, Proceedings Fourth International Conference on VLDB, 1978.
- [Blas79] M. Blasgen, J. Gray, M. Mitoma, T. Price, *The Convoy Phenomenon*, Operating Systems Review, Vol. 13, No. 2 (April 1979), 20-25.
- [Bora80] H. Boral, D. J. DeWitt, *Design Considerations for Data-flow Database Machines*, Proceedings of ACM-SIGMOD Conference on Management of Data, Orlando, 1980, 94-104.
- [Bora81] H. Boral, D. J. DeWitt, *Database Machine Activities at The University of Wisconsin*, Database Engineering, Vol. 4, No. 2 (Dec. 1981), 20-27.
- [Cana74] R. H. Canaday, R. D. Harrison, E. L. Ivie, J. L. Rydery, L. A. Wehr, *A Back-end Computer for Data Management*, CACM, Vol. 17, No. 10 (Oct. 1974), 575-582.
- [Cham74] D. D. Chamberlin, R. F. Boyce, *SEQUEL: A Structured English Query Language*, Proceedings ACM-SIGMOD Workshop on Data Description, Access, and Control, Ann Arbor, Michigan, May 1974, 249-264.
- [Chap80] G. A. Chapine, *Back-End Technology Trends*, IEEE Computer Magazine, Vol. 13, No. 2 (Feb. 1980), 50-58.

- [Chan78] H. Chang, *On Bubble Memories and Relational Data Bases*, Proceedings 4th International Conference on Very Large Data Bases, 1978, 207-229.
- [Chla80] I. Chlamtac, W. R. Franta, P. C. Patton, B. Wells, *Performance Issues in Back-End Storage Networks*, IEEE Computer Magazine, Vol. 13, No. 2 (Feb. 1980), 18-31.
- [CODA71] *CODASYL Data Base Task Group April 71 Report*, ACM, New York, 1971.
- [Codd70] E. F. Codd, *A Relational Model for Large Shared Data Banks*, CACM, Vol. 13, No. 6 (June, 1970), 377-387.
- [Cull78] *IDMS DML Programmer's Reference Guide*, Cullinane Corp., Wellesley, Mass., 1978.
- [Date75] C. J. Date, *An Introduction to Data Base Systems*, Addison-Wesley, Reading, Mass., 1975.
- [Date83] C. J. Date, *An Introduction to Data Base Systems*, Vol. II, Addison-Wesley, Reading, Mass., 1983.
- [DeFi73] C. R. DeFiore, P. B. Berra, *A Data Management System Utilising an Associative Memory*, Proceedings of the ACM National Computer Conference, 1973, 181-185.
- [DeWi78] D. DeWitt, *DIRECT - A Multiprocessor Organization for Supporting Relational Data Base Management Systems*, Proc. Fifth Annual Symposium on Computer Architecture, 1978, 182-189.
- [DeWi79] D. DeWitt, *Query Execution in DIRECT*, Proceedings of ACM-SIGMOD Conference on Management of Data, Boston, 1979, 13-22.
- [Dion80] J. Dion, *The Cambridge File Server*, Operating Systems Review, Vol. 14, No. 4 (Oct. 1980), 26-35.
- [Doga80] A. Dogac, E. A. Ozkarahan, *A Generalized DBMS Implementation on a Database Machine* Proceedings of ACM-SIGMOD 1980 International Conference on Management of Data, Santa Monica, California, May 1980, 133-143.
- [Emul80] *SC21/B1 SC21/BF SC21/V1 (RM02/RM09/RM05 Compatible) Disk Controller*, Emulex Corporation, 1980.
- [Epst80] R. Epstein, P. Hawthorn, *Design Decisions for the Intelligent Database Machine*, AFIPS Conference Proceedings, 1980 National Computer Conference, Vol. 49 (May 1980), 237-241.
- [Frid81] M. Fridrich, W. Older, *The FELIX File Server*, Proceedings of the Eighth Symposium on Operating Systems Principles, Dec., 1981 (also printed as Operating Systems Review, Vol. 15, No. 5), 37-44.
- [Gard81] G. Gardarin, *An Introduction to SABRE, A Multi Micro-Processor Data Base Machine*, Sixth Workshop on Computer Architecture for Non-Numeric Processing,

Hyeres, June 1981.

- [Gold74] R. Goldberg, R. Hassinger, *The Double Paging Anomaly*, Proceedings AFIPS 1974 NCC, Vol. 43, AFIPS Press, Montvale, N. J., 195-199.
- [Gray78] J. Gray, *Notes on Data Base Operating Systems*, IBM San Jose Research Report RJ2188, Feb. 1978.
- [Gray79] J. N. Gray, V. Watson, *A Shared Segment and Interprocess Communication Facility* IBM San Jose Research Report RJ2450 (a revision of RJ1579), Jan., 1979.
- [Hawt79] P. Hawthorn, *Evaluation and Enhancement of the Performance of Relational Database Management Systems*, Ph. D. Thesis, U. C. Berkeley, Report No. UCB/ERL M79/70, 1979.
- [Hawt82] P. Hawthorn, D. J. DeWitt, *Performance Analysis of Alternative Database Machine Architectures*, IEEE Transactions on Software Engineering, Vol. 8, No. 1 (Jan 1982), 61-75.
- [Hsia81] D. K. Hsiao, *The Laboratory for Database Systems Research at the Ohio State University*, Database Engineering, IEEE, Vol. 4, No. 2 (Dec. 1981), 14-19.
- [IBM81] *IBM 3380 Direct Access Storage Description and User's Guide*, IBM, Order Number GA26-1644-1, Second Edition, Dec. 1981.
- [Inte82] *iDBP DBMS Reference Manual*, Preliminary Release 3, Order Number 222100, Intel Corporation, Austin, Texas, Feb. 1982.
- [Joy81a] W. Joy, R. Fabry, *An Architecture for Interprocess Communication in UNIX*, Computer Systems Research Group at U. C. Berkeley memo (1981).
- [Joy81b] W. Joy, R. Fabry, *Proposals for enhancement of UNIX on the VAX*, Computer Systems Research Group at U. C. Berkeley memo July 21, 1981, revised Aug. 31, 1981.
- [Kap180] J. Kaplan, *Buffer Management Policies in a Database System*, M. S. Report, University of California, Berkeley, 1980.
- [Keen81] M. M. Keenan, *A Comparative Performance Evaluation of Database Management Systems*, M. S. Report, Computer Science Division, University of California, Berkeley, Oct. 1981 (also Lawrence Berkeley Laboratory report LBL-13700).
- [Kiy081] Y. Kiyoki, K. Tanaka, H. Aiso, N. Kamibayashi, *Design and Evaluation of a Relational Data Base Machine Employing Advanced Data Structures and Algorithms* Proceedings of The 8th Annual Symposium on Computer Architecture, (SIGARCH Vol. 9, No. 3, May, 1981), 389-406.
- [Kung80] H. T. Kung, P. L. Lehman, *Systolic (VLSI) Arrays for Relational Database Operations*, Proceedings of ACM-SIGMOD Conference on Management of Data, Santa Monica, 1980, 105-116.

- [Lamp81] B. W. Lampson, *Distributed Systems -- Architecture and Implementation: An Advanced Course, Lecture Notes in Computer Science* B. W. Lampson, M. Paul, and H. J. Siegert ed., Springer-Verlag, 1981, 365-370.
- [Leil78] H. O. Leilich, G. Stiege, H. Ch. Zeidler, *A Search Processor for Data Base Management Systems*, Proceedings of the Fourth VLDB, Sept. 1978, 280-287.
- [Lin76] C. S. Lin, D. C. P. Smith, J. M. Smith, *The Design of a Rotating Associative Memory for Relational Data Base Applications*, ACM Transactions on Database Systems, Vol. 1, No.1, (March 1976), 53-65.
- [Lind79] B. G. Lindsay, P. G. Selinger, C. Galtier, J. N. Gray, R. A. Lorie, T. G. Price, F. Putzolu, I. L. Traiger, B. W. Wade, *Notes on Distributed Databases*, IBM San Jose Research Report RJ2571, 1979.
- [Lowe80] E. Lowenthal, *Database Management Systems for Local Area Networks*, IEEE Computer Magazine, August, 1982.
- [Mall79] V. A. J. Maller, *The Content Addressable File Store - CAFS*, ICL Technical Journal, Nov. 1979, 265-279.
- [Mari75] T. Marill, D. Stern, *The Datacomputer - a Network Utility*, AFIPS Conference Proceedings, 1975 NCC, Vol. 44, 389-395.
- [Metc76] R. M. Metcalfe, D. R. Boggs, *Ethernet: Distributed Packet Switching for Local Computer Networks*, CACM Vol. 19, No. 7 (July 1976), 395-404.
- [Mitc82] J. G. Mitchell, J. Dion, *A Comparison of Two Network-Based File Servers*, CACM, Vol. 25, No. 4 (April 1982), 233-245.
- [Nels81] B. J. Nelson, *Remote Procedure Call*, Ph.D. dissertation, Computer Science Department, Carnegie-Mellon University, CMU report number CMU-CS-81-119, Xerox PARC report number CSL-81-9, 1981.
- [Ofia] K. Ofiazer, *A Reconfigurable VLSI Architecture for a Database Processor*, unpublished paper available from the author at CMU.
- [Oliv79] E. J. Oliver, *RELACS, An Associative Computer Architecture to Support a Relational Data Model*, Ph. D. Thesis, Syracuse University, 1979.
- [Ozka75] E. Ozkaran, S. Schuster, K. Smith, *RAP - Associative Processor for Database Management*, AFIPS Conference Proceedings, 1975 NCC, Vol. 44, 379-388.
- [Ozka77] E. A. Ozkaran, S. A. Schuster, K. C. Sevcik, *Performance Evaluation of a Relational Associative Processor*, ACM Transactions on Database Systems, Vol. 2, No. 2, (June, 1977), 175-195.
- [Post80a] J. Postel ed., *DOD Standard Internet Protocol*, Internet Working Group, IEN 128, Jan. 1980.

- [Post80b] J. Postel ed., *DOD Standard Transmission Control Protocol*, Internet Working Group, IEN 129, Jan. 1980.
- [Ritc78] D. M. Ritchie, K. Thompson, *The UNIX Time-sharing System*, Bell System Technical Journal, Vol. 57, No. 6, (July-Aug. 1978), 1905-1930.
- [Seki83] A. Sekino, K. Takeuchi, T. Goto, K. Hara, *Design and Implementation of an Information Query Computer*, Proceedings of Compcon, San Francisco, 1983, 374-377.
- [Seo81] K. Seo, H. Aiso, N. Kamibayashi, *A Look-Ahead Data Staging Architecture for Relational Data Base Machines*, Proceedings of The 8th Annual Symposium on Computer Architecture, (SIGARCH Vol. 9, No. 3, May, 1981), 389-406.
- [Shaw80] D. E. Shaw, *A Relational Database Machine Architecture*, Fifth Workshop on Computer Architecture for Non-Numeric Processing, March, 1980, 84-95.
- [Shaw82] D. E. Shaw, S. J. Stolfo, H. Ibrahim, B. Hillyer, G. Wiederhold, J. A. Andrews *The NON-VON Database Machine: A Brief Overview*, Database Engineering, IEEE, Vol. 4, No. 2 (Dec. 1981), 28-30.
- [Shib82] S. Shibayama, T. Kakuta, N. Miyazaki, H. Yokota, K. Murakami, *A Relational Database Machine "Delta"*, Technical Memorandum of ICOT Research Center No. TM-003, Nov. 1982.
- [Soft] *The Database Machine*, a promotional brochure published by Software AG of North America, Inc.
- [Soft81] *ADABAS Channel-to Channel Software System Installation and Operations Manual*, Software AG of North America, Inc., 1981.
- [Spec82] A. Z. Spector, *Performing Remote Operations Efficiently on a Local Computer Network*, CACM, Vol. 25, No. 4 (April 1982), 246-259.
- [Ston76] M. Stonebraker, E. Wong, P. Kreps, G. Held, *The Design and Implementation of INGRES*, ACM Transactions on Database Systems, Vol. 1, No. 3 (Sept., 1976) 189-222.
- [Ston80] M. Stonebraker, *Retrospection on a Database System*, ACM Transactions on Database Systems, Vol. 5, No. 2 (June 1980), 225-240.
- [Ston81] M. Stonebraker, *Operating System Support for Database Management* CACM, Vol. 24, No. 7 (July 1981), 412-418.
- [Su75] S. Su, G. Lipovski, *CASSM: A Cellular System for Very Large Data Bases*, Proceedings of the International Conference on Very Large Data Bases, Framingham, Massachusetts, 1975, 456-472.
- [Swin79] D. Swinehart, G. McDaniel, D. Boggs, *WFS: A Simple Shared File System for a Distributed Environment*, Proceedings of the Seventh Symposium on Operating System Principles, Asilomar, California, Dec. 1979, 9-17.

- [Thor80] J. E. Thornton, *Back-End Network Approaches*, IEEE Computer Magazine, Vol. 13, No. 2 (Feb. 1980), 10-17.
- [Trai82] I. L. Traiger, *Virtual Memory Management for Data Base Systems*, Operating Systems Review, Vol. 16, No. 4 (Oct. 1982), 26-48.
- [Ubel82] M. Ubell, *The Intelligent Database Machine*, Database Engineering, IEEE, Vol. 4, No. 2 (Dec. 1981), 28-30.
- [Uemu83] S. Uemura, *Database Machine Activities in Japan*, Database Engineering, IEEE, Vol. 6, No. 1 (March 1983), 63-64.
- [Ullm80] J. D. Ullman, *Principles of Database Systems*, Computer Science Press, Potomac, Maryland, 1980.
- [Vald82] P. Valduriez, *Semi-join Algorithms for Multi-processor Systems*, Proceedings of ACM-SIGMOD Conference on Management of Data, Orlando, 1982, 225-233.
- [Wah80] B. W. Wah, S. B. Yao, *DIALOG -- A Distributed Processor Organization for Database Machines*, AFIPS National Computer Conference, Vol. 49, 1980, 243-255.
- [Wats80] R. W. Watson, *Network Architecture Design for Back-End Storage Networks*, IEEE Computer Magazine, Vol. 13, No. 2 (Feb. 1980), 32-48.