A DYNAMIC PRIMED-PROCESS POOL FOR THE VOLCANO QUERY EVALUATION SYSTEM

Gerald Kenneth Bortvedt
B.S., Washington State University, 1978

A thesis submitted to the faculty
of the Oregon Graduate Center
in partial fulfillment of the
requirements for the degree
Master of Science
in
Computer Science

June, 1989

The thesis "A Dynamic Primed-Process Pool for the Volcano Query Evaluation System" by Gerald K. Bortvedt has been examined and approved by the following Examination Committee:

Goetz Graefe
Assistant Professor, Oregon Graduate Center
Thesis advisor

David Maier
Professor, Oregon Graduate Center

Leonard Shapiro
Chair Computer Science Department,
Portland State University

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank my thesis committee, Dr. Goetz Graefe, Dr. David Maier and Dr. Leonard Shapiro for reviewing my thesis. Their suggestions greatly improved the quality of this thesis. I give special thanks to Dr. Goetz Graefe, my thesis advisor. His encouragement and guidance was instrumental in my success at the Oregon Graduate Center. Most of all I would like to thank my wife, Mary, for her infinite patience and support.

ABSTRACT

Implementation of a Dynamic Primed Process Pool
for the Volcano Query Evaluation System

Gerald K. Bortvedt, M.S.
Oregon Graduate Center, 1989

Supervising Professor: Goetz Graefe

Volcano is a single-user, high-performance query evaluation system, capable of parallel query evaluation on a shared-memory multi-processor machine such as the Sequent Symmetry. Ancillary processes are forked by each query as needed using the UNIX "fork" system call. The cost of forking a process can be very high and has proven to be a detriment to performance. As a solution to this problem a system of persistent, reusable processes, a "primed-process pool", is implemented. Mechanisms for dynamic process loading, interprocess communication and dynamic process creation are explored and their implementations are discussed. An experimental performance evaluation is done comparing the primed-process pool with the existing system. The mechanisms required to extend the primed-process pool to a network of shared-memory machines are explored.

CHAPTER 1

INTRODUCTION

The increased availability of multi-processor computers has focused attention on the advantages and problems of parallel programming. The database world is no exception. Within the relational database area considerable effort has been spent to exploit this technology. The Gamma [1] project at University of Wisconsin, the Bubba [2] project at MCC, the Grace [3] database machine at the University of Tokyo, XPRS [4] at University of California at Berkeley, Teradata, [5] at the Teradata corporation and the Volcano [6] query evaluation system at The Oregon Graduate Center, are a few examples.

To get a program to execute in parallel requires mechanisms to coordinate the various processes involved. These mechanisms can add significantly to the cost of executing the program. A very important aspect of parallel programming is to minimize these costs. How and when new processes are created is one mechanism that needs to be explored. The cost of creating new processes can be very high. In an environment that continually requires a group of similar processes, i.e., a query evaluation system, it is beneficial to allow processes to exist between uses in an idle state, so they can be reused at a later time. These benefits must, however, be weighed against the cost of maintaining and communicating with these processes.

In this thesis we explore the implementation of a pool of reusable processes for the Volcano query evaluation system. The performance of the system is compared experimentally with the current system to see if indeed the benefits outweigh the costs.

1.1. Motivation

In the current implementation of Volcano, a query is evaluated by compiling it with the Volcano library and then executing the resulting program. The Volcano library contains functions necessary to evaluate a query. If parallel execution is required, the process creates copies of itself using the Dynix[1] "fork" system call. The cost of the Dynix "fork" system call can add significantly to the overhead of parallel query evaluation.

When a fork call is executed, the operating system must allocate a new process structure and copy into it all the information maintained in

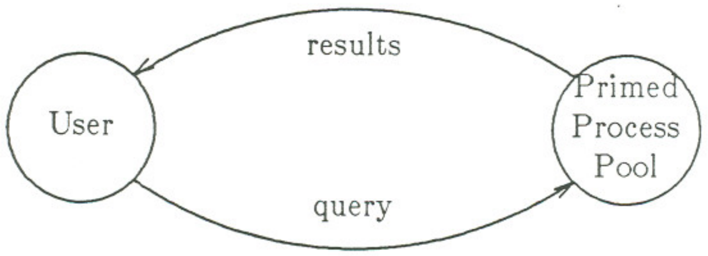
[1]Dynix is a trade mark of the Sequent Corporation, it is an enhanced version of the UNIX operating system.

the original process structure[7]. This information includes such things as the process scheduling priority, pointers to other control blocks and lists of related processes. A new page table is constructed and space in memory is allocated for the new process. The data and stack segments of the original process are then copied into this space. The text segment can be shared so there is no reason to copy it. Copying of the data segment, stack segments, and page table accounts for much of the cost of the "fork" system call. This cost increases with the size of the process being forked and the amount of space it has allocated.

With the current implementation of Volcano new processes are created for every query. When the evaluation of the query is complete the processes are terminated. The processes used to evaluate a particular query contains some subset of the routines contained in the Volcano library. If a group of processes containing the entire Volcano library existed and persisted from one query to the next, forking new processes for each query would be unnecessary. The cost of creating new processes would be paid once rather than for each query. The cost of forking processes could be amortized over an arbitrary number of queries rather than paid by each query individually. Reusing processes rather than creating new ones could reduce the execution cost of each query.

One goal of this project was to develop and implement the mechanisms necessary for query evaluator processes to be reused, eliminating the need to fork new processes for each query. This required the processes used to evaluate the query to exist independent of the queries themselves. The evaluator processes exist as a group of background processes waiting for a request to evaluate a query. This group of processes will be referred to as the *primed-process pool.* The primed-process pool will require a series of mechanisms that can take a query input in some form, send it, directly or indirectly, to the evaluator processes (the primed-process pool), convert the query into a form the evaluators understand, wait for the query to be evaluated, and then return the results to the user (Figure 1). Furthermore, a mechanism must be developed to activate processes within the pool as they are needed by a single query for parallel evaluation.

A second goal of this project is to explore designs and mechanisms necessary to extend the primed-process pool and thus Volcano over machine boundaries. Volcano is currently implemented to execute on a multi-processor shared-memory machine. It is the belief of the Volcano designers that high performance can be achieved by a network of shared-memory machines[8]. Extending the primed-process pool entails developing new or extending the existing communication mechanisms to work between



A Primed-Process Pool
Figure 1

machines. The data records need to be collected and sent between processes on different machines rather than just pointers, as can be done in a shared-memory environment. We would like to maintain the encapsulation of the parallelism that currently exists in Volcano[6]. The fact that the query processing is being done on multiple machines should be transparent to the user and all other procedures within the Volcano system, i.e, no module within Volcano other than the *exchange module* (the module responsible for parallelism) should have to treat records from a remote machine in any special way.

1.2. Structure of the Thesis

In the following chapter we discuss other work in this area. A brief description of the Volcano query evaluation system follows, with emphasis on the parts that are directly affected by this project. Chapter 4 discusses the design and implementation of a reusable primed-process pool for Volcano. Chapter 5 presents experimental performance comparisons between the forkint system and the primed-process pool. The final chapter contains a summary of the project, conclusions that can be drawn from the results and what direction should be pursued next.

CHAPTER 2
RELATED WORK

A number of corporations and universities have developed high performance, multi-processor database machines. They fall into two basic categories, the shared-nothing machines, i.e., Gamma[1] and Bubba[2], and the shared-memory machines, i.e., Direct[9], Grace[3] and XPRS[4]. Of these, only Direct and Gamma directly address the implementation issues discussed in this thesis. Brief descriptions of the others are included.

XPRS is being developed at the University of California, Berkeley. The goal of XPRS is to build a high performance database system that will run on a conventional operating system[4]. It is designed to run on a single processor machine, e.g., a Sun 4, or a multi-processor shared-memory machine, e.g., a Sequent Symmetry. XPRS is being designed to take advantage of expected advances in disk technology, bus speed and memory size and speed.

Grace [3] is a parallel relational database machine being developed at the University of Tokyo. Grace aims to achieve high performance by using *data-stream-oriented processing.* In data-stream processing each relational operator is encapsulated in a module. The operand data or *tasks* are passed between modules one task at a time. A similar system is used in Volcano.

Bubba [2] is a highly parallel, shared-nothing database machine developed at MCC in Austin, Texas. It is designed to employ on the order of 1000 *intelligent repositories* (IRs). An intelligent repository consists of a microprocessor, a disk controller, communication processor, a large main memory and a disk. The IRs are connected by a network so any IR can communicate to any other IR. The IRs are physically close to keep message delays small.

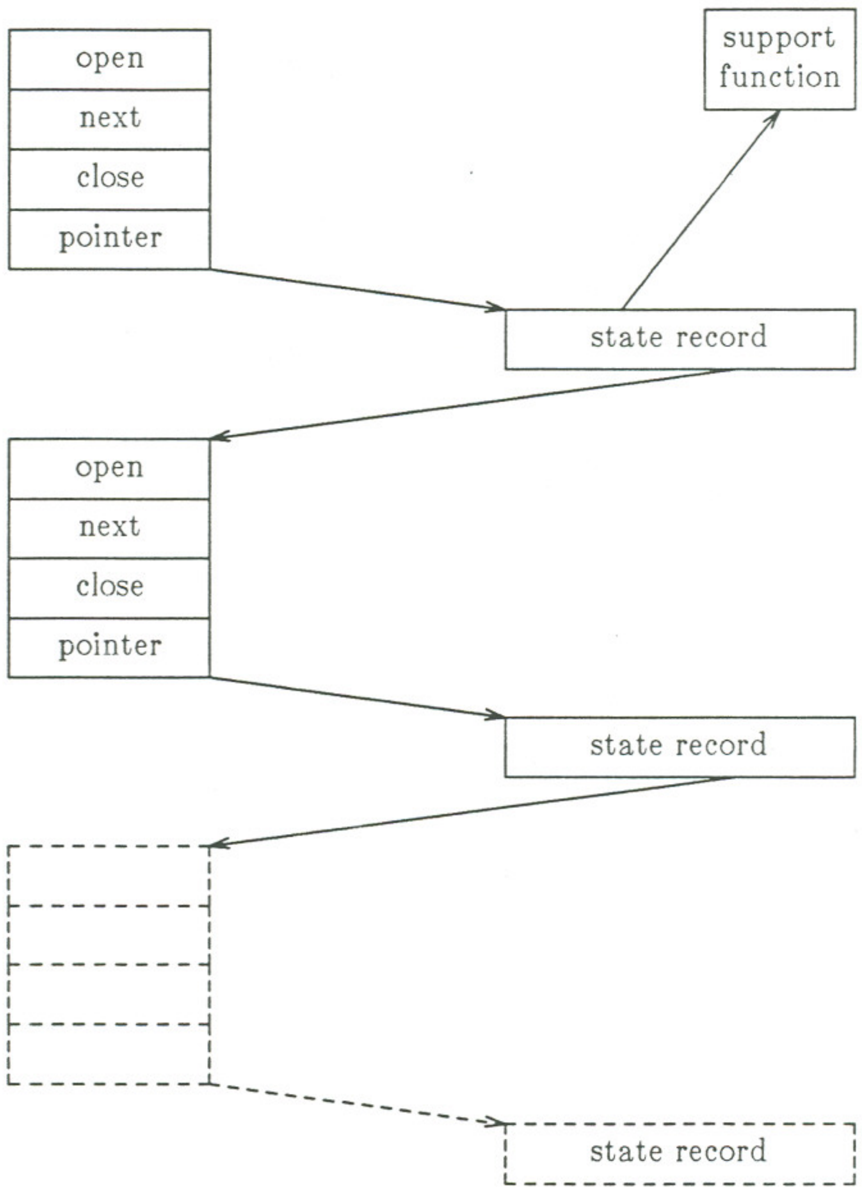
Direct [9] is a multi-user database machine using the Version 6 UNIX operating system. The *host* is a PDP 11/40, the back-end consists of eight LSI 11/23 computers. The software protocol used to connect the front-end host processes to the back-end (the server processes) is the UNIX port facility. A port is an interprocess communication facility that allows a many-to-one connection. The port is "owned" by a single process, that process has exclusive read access to the port. Any number of processes can write to it. The port allows for multiple users on the host to communicate with the single controller process on the back-end. The back-end software consists of four main process types, a process to manage the catalogs, a process to manage memory, a process to schedule a query's execution and the query evaluator processes. The first three processes are implemented on one processor. The remaining processors are used by query evaluator processes, one per processor. To avoid writing a separate communication subsystem, the port facility is also used for the interprocess communication among back-end processes.

Gamma [1] is an operational prototype of a shared-nothing database machine. It consists of 17 VAX 11/750 processors each with two megabytes of memory. The processors are connected with a token ring. The host, another VAX 11/750, is also included in the token ring. The host machine uses the Berkeley UNIX operating system with communication facilities similar to those in Version 6 UNIX. However, the database machine itself uses NOSE, a minimal operating system developed to support database management systems. Nose provides a fast, reliable communication link between the server processes and to the host machine. Gamma uses a primed-process system. There are six distinct process types: catalog manager, query manager, scheduler process, operator process, deadlock detection process and a log manager. The catalog manager and the query managers (one for each user) reside on the host machine. The other processes reside on the 17 processor back-end machine. There is a scheduler process for each query. All the scheduler processes were implemented on a single processor. There is a single log manager and a single deadlock detection process for the entire system, each are implemented on a separate processor. The remaining processors are used by operator processes. NOSE and Gamma are currently being ported to an Intel IPSC/2 hypercube with 32 nodes and 32 disks.

CHAPTER 3
VOLCANO

Volcano is a high performance query evaluation system developed for database research and education[8]. It is designed as a set of mechanisms for exploring database design policies, such as how and when to use parallel processing. Volcano is currently a single-user system capable of parallel query evaluation. The system is based on a small core of operator modules (about 15), e.g. a sort module, a file scan module, a module to create indices. All the operators are implemented as *iterators,* i.e., each consists of an open, next, and close function. The operators are supplemented by *support functions.* Support functions are specific to particular types of data records, e.g., comparisons and hash functions. They are passed to the operators by means of pointers.

To evaluate a query, a tree-structured query evaluation plan is generated by the user (Figure 2). Each node in the tree consists of four pointers. The first three pointers are to the open, next and close functions of the operator associated with that node. The fourth is a pointer to a *state record* which contains the arguments necessary for that particular



Query Evaluation Tree
Figure 2

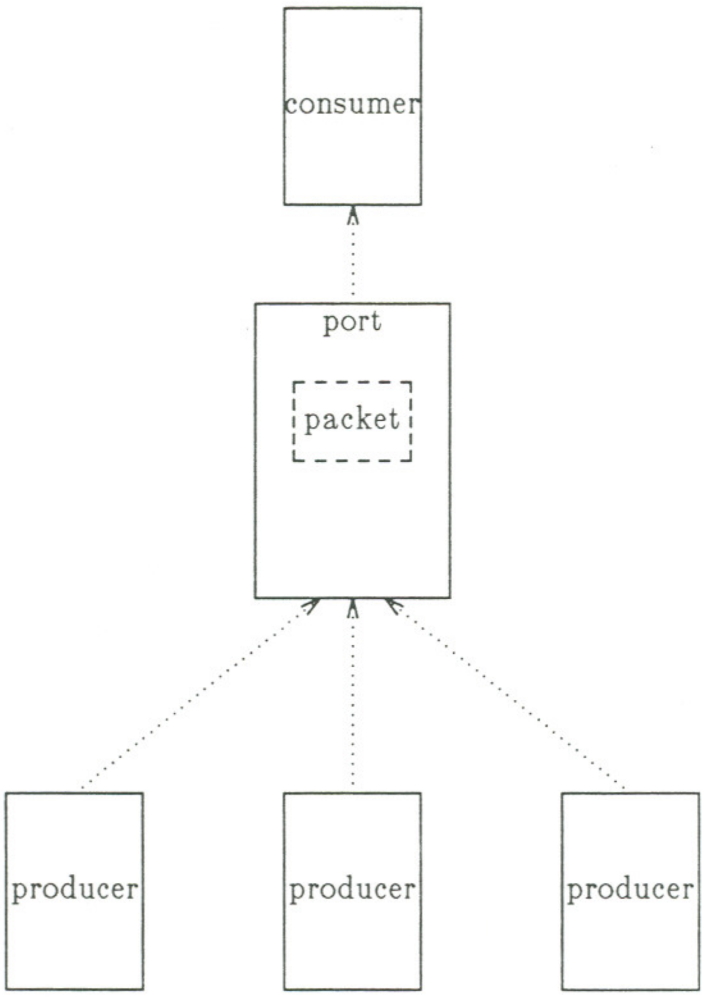
operator, such as pointers to support functions, and a pointer to the next node or nodes in the tree. All the information associated with each node in the tree (the state) is also kept in its state record. Encapsulating this information in the state record allows an operator to be used an arbitrary number of times in a single query. A query tree can be arbitrarily complex and the operators can be arranged in any logical manner.

To pass information from one operator to another a *NEXT_RECORD* structure is used. A NEXT_RECORD structure consists of the record identifier (its physical location on the disk) and a pointer to the location in the buffer where the record currently resides. Passing pointers rather than the records themselves eliminates the need for memory-to-memory copying of records and contributes to Volcano's efficiency. To pass a NEXT_RECORD structure rather than the record itself requires the record to be fixed in the buffer until all references to it are no longer required. For more information on the Volcano's buffer system see reference[8].

Volcano's parallelism is encapsulated in a single module, the exchange module[6]. The exchange module follows the same open, next, close paradigm as all the other operators in the system. This uniformity allows the exchange module to be inserted anywhere in a query tree. The exchange module uses the Dynix "fork" system call to create processes as needed. To decrease the time required the processes are forked using a *propagation scheme.* The first process forks a process then both these processes fork new processes and so on until the required number of processes have been forked.

When the *open_exchange* function is invoked it creates a *port.* A port is a data structure in shared memory used to pass records from one process to another. The open_exchange then forks a single process. The original process is now a consumer of the records that the new process, the producer, produces. This producer process, *the master producer,* immediately executes the function *execute_producers.* This function forks the remaining producers, if any more are required. Each producer starts executing the query tree. The producers, rather than passing the records one by one to the calling operator, as the other operators do, collect the NEXT_RECORD structures in *packets* (Figure 3). When a packet is full, it is inserted into the appropriate port and the consumer is alerted, by means of a semaphore, that data is available. The consumer executes the next_exchange function which returns the records to the requesting operator one record at-a-time.

The exchange module can be used for both intra-operator parallelism, i.e., the work of one operator spread out over a number of processes, and inter-operator parallelism, i.e., different operators executed by different processes.



The Exchange Module
Figure 3

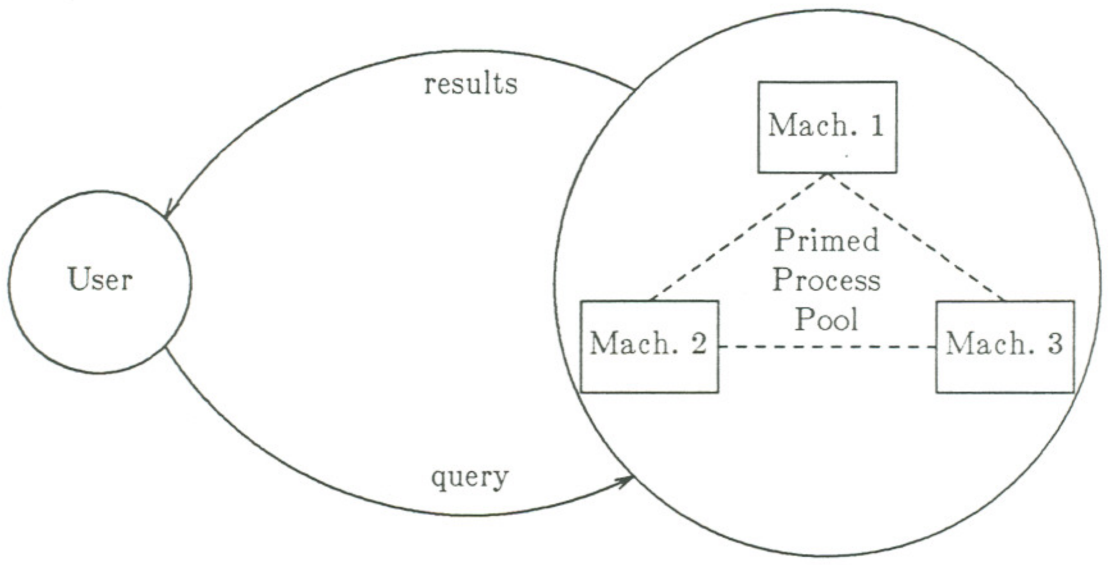
CHAPTER 4

THE PRIMED-PROCESS SYSTEM

The primed-process system for Volcano was developed in five stages. These stages are examined in the five sections of this chapter. The first section addresses the need to separate the query from the query evaluator and how to associate them at run time. The next section explores the communication requirements of the primed process system using only a single evaluator process. The extension of this system to multiple evaluators is presented in Section 3. Section 4 deals with the problem of a fixed size process pool and demonstrates the solution. In the final section we explore the needs and present a design for extending the primed-process pool over a network of shared-memory machines (Figure 4).

4.1. Incremental Loading

In this section we explore a mechanism that allows a compiled program to be associated with another program that already exists as an executing process. This procedure, known as dynamic or incremental loading, is provided by the Dynix link editor.

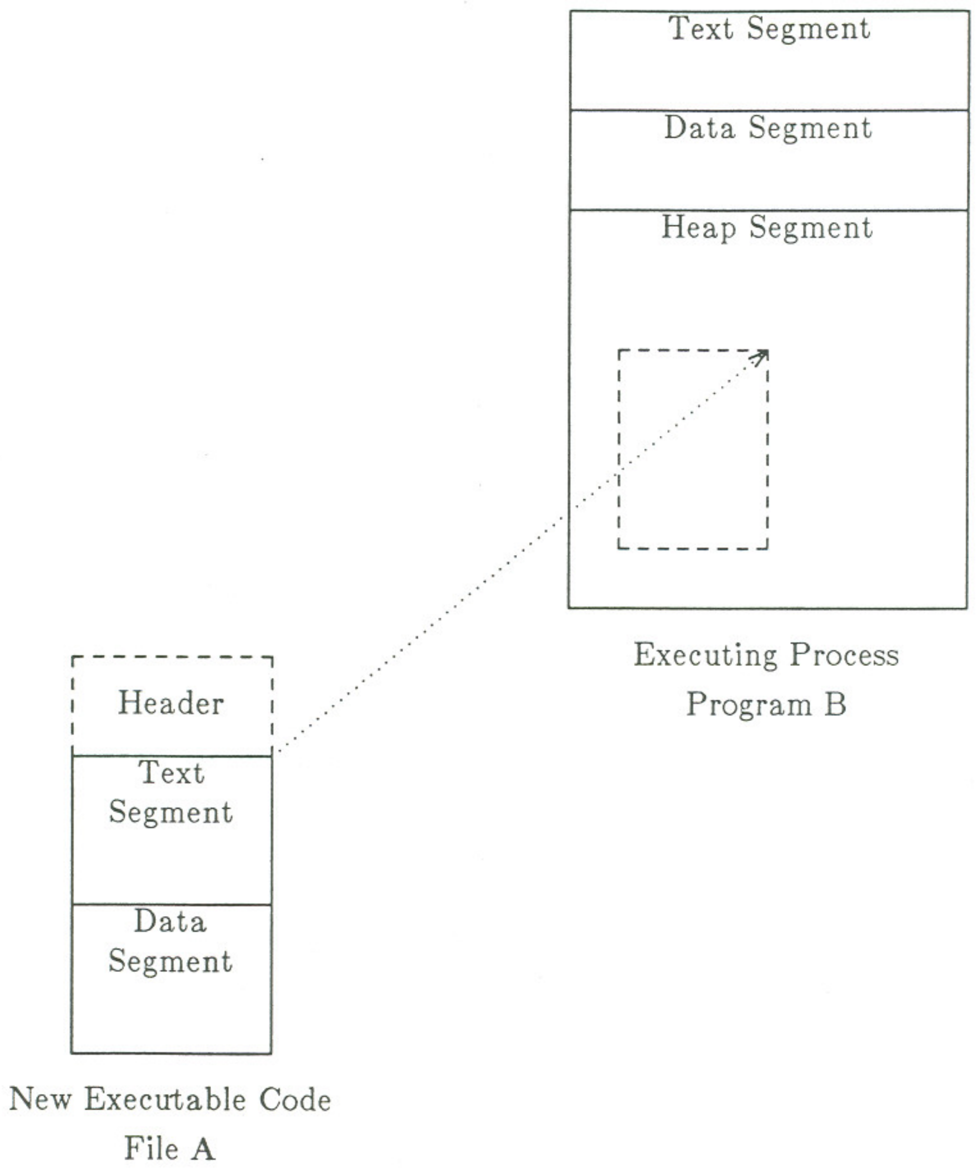


A Distributed Primed-Process Pool
Figure 4

In the existing Volcano system, a query is compiled together with the Volcano library to create an executable program. This program is only capable of evaluating one particular query. A new executable program must be created for each query to be evaluated. It is preferable to have one executable program that can evaluate any query. For a program to be capable of evaluating any query, the program must exist independently of the queries it evaluates. To evaluate the query, the query is associated with the query evaluator at run time. If a query were a simple data structure, associating the query with the query evaluator would be very straightforward. The query could be read into the process directly. Queries, however, include compiled code that must be linked to the evaluator process. To link the query and the evaluator process requires a mechanism that allows new code, the query, to be inserted into a running process, the query evaluator, in such a way that the running process can execute the new code. Once the query is evaluated, the code associated with it should disappear, leaving the query evaluator ready to accept another query. The incremental loading facilities of the link editor achieves both of these goals.

Incremental loading can be done by using the link editor (ld) command with the -A option. The result of this command is the creation of an executable file, (file A in Figure 5) that may be read into an existing program's space, (program B in Figure 5) and executed as an extension to the existing program. The symbol table of the newly linked file, file A will reflect the entries of both the object file being linked and those of the existing program, program B. The symbol table of the currently executing program (program B) remains unchanged. The ld command takes as arguments: the name of the executable program (program B) the file will be read into, the location within the address space of program B where the new executable code (file A) will be placed and the point within this new code (the text segment of file A) where execution will begin (the entry point). Sufficient space must be allocated in the data space of program B, at the location specified, to accommodate the text and data segments of the new object code, file A. The size of the data and text segments are available in the header structure of the object file. This space can either be preallocated in the data segment or it may be dynamically allocated in the heap space of the existing process (Figure 5). The code is then read into this location by the executing program (program B) using the "read" system call. Both the space allocation and the read are explicitly coded in program B, they are not part of the link editor command. Once the text and data segments from file A have been read into the data space of program B, program B can access both the text and data. The text segment of file A is accessed by program B using a function call. The function is addressed by using the entry point specified in the ld command. The address of the entry point is read from the header structure of file A. The new text segment read in from file A is not write protected as would normally be the case with statically loaded text. It is thus possible to overwrite the executable code.

When the system is extended to a multi-process system (the primed-process pool), each process will need to access the code and data that is dynamically loaded. To avoid reloading the query into each process, the query is read into shared-memory. Any process within the process pool can directly access both the code and the data associated with the query. The



Incremental Load into Heap Space
Figure 5

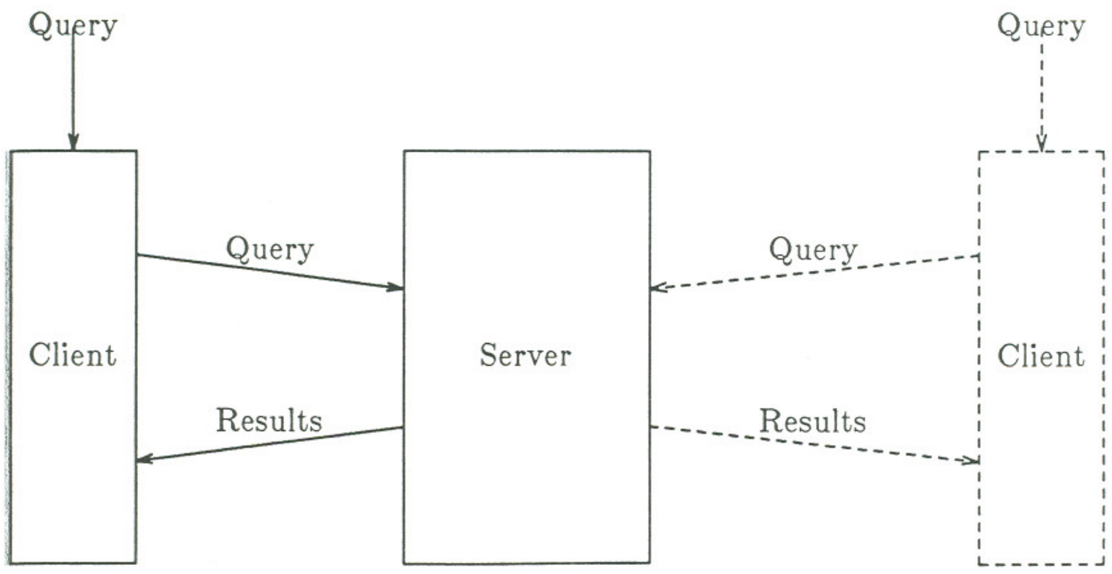
consequences of having the query's data space in shared-memory will be discussed later.

It was considered desirable to change both the Volcano library and the current test queries as little as possible. With multiple people working on the same code concurrently, the fewer changes necessary, the fewer problems coalescing the changes and the less chance of introducing undesired side effects. While the existing queries are for test purposes only, if they can be used without major changes, valid performance comparisons will be easier to make. The only change necessary in the test queries was to change the name of the query function from "main" to the name of the file the function was stored in, e.g., if the file was called "query1" then the function was also called "query1". This change was necessary to help establish the entry point of the function in the newly loaded code. This phase required no changes to the Volcano code itself.

4.2. Client/Server

The incremental loading facilities of the link editor allows us to separate the query from the query evaluator. The query evaluator still exists as a foreground process and, therefore, can only exist for one query evaluation session. In the current implementation, if some number of queries are evaluated, then the terminal is needed for some unrelated work, the query evaluator has to be terminated. When it is required again it has to be restarted. We prefer the evaluator process existed as an idle background process between sessions, to be activated as necessary. This mechanism is a basic client/server apparatus[10] (Figure 6). We will now discuss the design and implementation of this apparatus for Volcano.

The client in this case is a process that runs as a foreground process. It is capable of accepting input from *stdin* and places its output on *stdout* or *stderr*. It exists independently of the query and the query evaluator (the server). It is possible to have multiple clients that are independent of each



The Client/Server Apparatus
Figure 6

other (This property is of limited usefulness at this time since Volcano is a single-user system with no concurrency control. It is, however, a prerequisite to extending it to a multi-user system and, therefore, was included.). At this stage, the purpose of the client process is to accept a query from the input device, relay it to the query evaluator and wait for the results to be returned. The client exists for the duration of a query evaluation session, which can consist of an arbitrary number of queries.

The server or query evaluator is a process (or later a group of processes) that exists in the background, independently of any terminal or other input device. The server has no inherent relationship with any particular client or query. The server contains all the code necessary to evaluate any legal query (not including query specific support functions), i.e., the entire Volcano library. It also contains the code necessary to dynamically load a query as discussed in the previous section. The query is received from the particular client being served, loaded, evaluated and the results are returned to the client. The server is then ready to evaluate another query from the same or any other client. The server is totally independent of the clients. It persists until it is explicitly killed, regardless of the existence (or lack of) client processes.

A major part of any client/server apparatus is the communication scheme used to connect the client with the server. The Dynix operating system, being implemented on a multi-process shared-memory machine, offers a number of possible schemes for this communication. A location in shared-memory can serve for message passing, the memory-mapping facility may be used, or the data may be passed using sockets, an interprocess communication facility provided by the operating system. Dynix also supports the Unix system V message-passing facility. These alternatives are discussed briefly.

Using shared-memory to pass messages between two processes is very straightforward. One process puts a message into the space, the other takes it out. The problem is how to efficiently detect when a message is available. If the receiving process must continually check to see if there is a message available, it is in essence in a state of busy wait. Busy waits and high performance are not normally compatible. The Dynix operating system has a signal facility that would allow the sender to alert the receiver that there is a message available, but this scheme requires the sender to know the receiver's process identification number. To establish a connection the processes involved also need to know the location in shared-memory where the messages reside. To obtain this information the processes must communicate before the communication link can be created. This information can be shared by storing it in a file known to both the clients and the server.

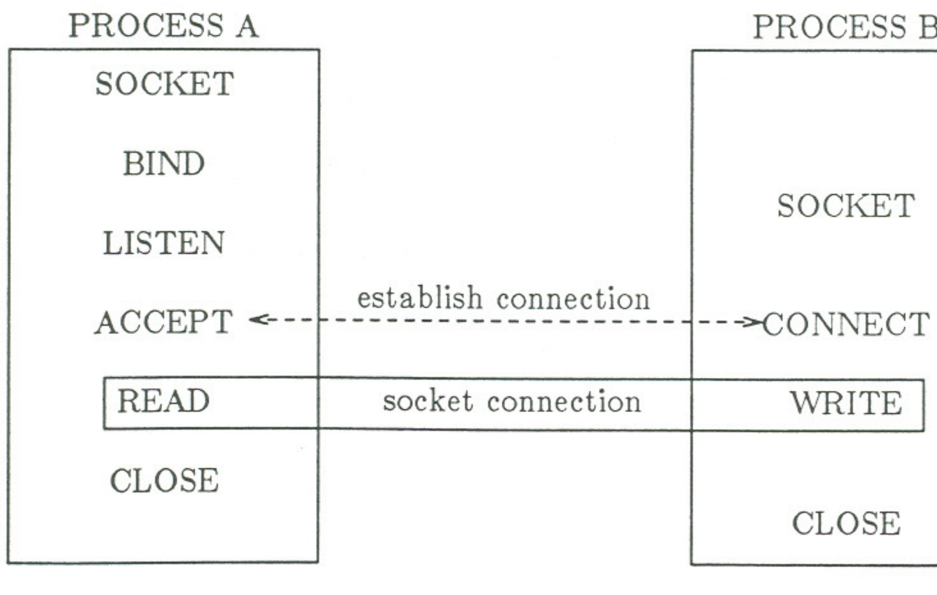
There is also a semaphore mechanism available to synchronize shared-memory access. This mechanism is more effective than signals to manage the shared-space and alert processes to the presence of messages. Using semaphores still requires information to be shared between the processes, the location in shared-memory and the location of the semaphore, before the communication link is established.

The memory-mapping facility allows a file to be mapped into the data space of a process. This facility does not require a special file, as the previous schemes do, to hold information necessary to establish a connection. If both the client and the server know the name of the file being mapped the connection can be established. The information needed to create the control structure, either signals or semaphores, can then be passed. The major shortcoming of this idea is its complexity. The mechanisms for alerting a process a message is available and that used to actually pass the message are separate. One mechanism that does both is preferable.

The socket facility seems to be the most appropriate of the mechanisms examined thus far, for this application. Using sockets is fairly complex and deserves some explanation. Sockets come in various flavors depending on the environment they are being used in and the reliability required. The variety we used were of the style SOCK_STREAM in the UNIX domain. This variety was chosen based on the available environment (a UNIX type environment) and for reliability. The other type of sockets currently implemented in the UNIX domain are of the style DATAGRAM, which are not guaranteed to be delivered and thus require a "message received" response[11]. Since message passing of any kind is costly we wish to avoid the need for the response message.

With a SOCK_STREAM type socket communication takes place across a connection between two processes. The connection is established by a series of system calls (Figure 7). Process A uses the system call "socket" to create a socket of a specified type. The call returns a *file descriptor*, an integer that is an index into process A's open file table. The socket call establishes a *socket*, one end of the communication connection. The system call "bind" is then used to bind a file name to the socket. The system call "listen" is used to establish that this socket is ready to accept a connection and to set the size of the message queue. Currently the maximum number of messages the operating system will queue is five. The process can now accept a connection using the system call "accept". The process will block on this call until a connection is established. When a connection is established, the "accept" call returns another file descriptor that can then be used to read or write a message across the communication link using the "read" or "write" system call respectively.

For process B (Figure 7) to connect with process A's waiting socket, process B also uses the "socket" system call, which again returns a file descriptor and establishes another socket. Process B uses the "connect" system call, with the socket name bound by process A as an argument, to establish the connection between the sockets in process A and process B. The socket name must be known by both process A and process B. If the



System Calls Needed to Create and Connect a Socket
Figure 7

connect is successful, process B can read or write messages to process A, and visa versa, using the "read" or "write" system call respectively, with the file descriptor returned by the "socket" call. The connection can be terminated by either process using the "close" system call. To re-establish the connection both processes must close their respective connections, then execute the system calls as before. Up to 4096 bytes can be buffered by the socket. One possible hazard is that message boundaries are not preserved. If there is more than one message queued in the socket buffer, care must be taken that the length of the message read is equal to the length of the message written or the messages may not arrive in the intended format.

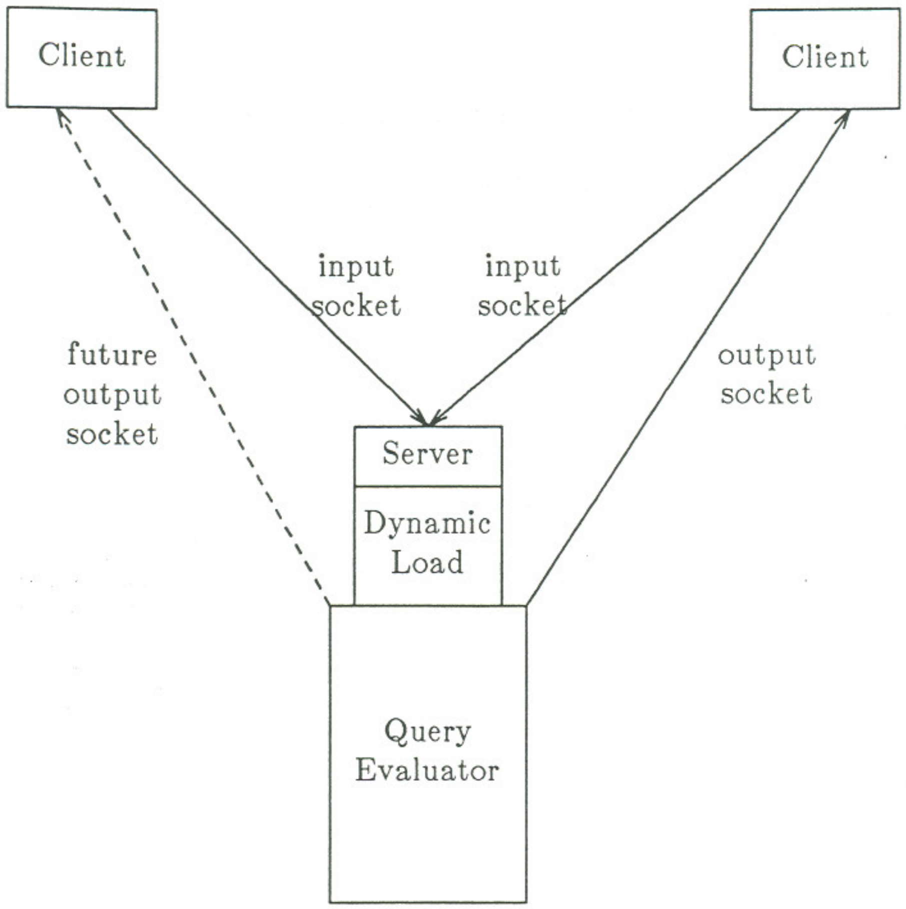
The Unix system V message facilities offer features very similar to sockets. Their implementation is, however, quite different. The structure used to pass messages, unlike sockets, is not part of the operating system's file system. This message facility builds a separate queue structure. Building a separate queue structure has the advantage that it does not use the process's open file table (a very limited resource on the available machine). However, it prohibits the easy redirection of data that is possible with sockets. This message system is not extendible over machine boundaries. Since data redirection and extension of the system over machine boundaries are important in this project, we have used sockets in our implementation.

In our implementation (Figure 8), the server is started as a background process. It immediately creates a socket whose name is known to the client and then blocks on the "accept" system call waiting for a client to connect. The socket name is a compile time constant for both the clients and the server. When a client is started as a foreground process (Figure 8), it creates a unique file name, whose use will be discussed

Server	**Client**
Starts in Background	
Create Socket	
	Start in Foreground
	Create Unique File Name
	Read in Query
	Establish Socket Connection with Server
	Send Query and Unique File Name
	Close First Socket, Create New Socket
Establish new Socket Connect with Client	
Redirect Output Through New Socket	
Dynamically Load Query	
Evaluate Query	
Returns Results to Client	Return Results to the User
Waits for Client to Connect	Wait for New Query

Order of Execution for Server and Client Processes
Figure 8

shortly, then blocks on a read from the keyboard. When data is entered on the keyboard, presumably the name of a file containing a query along with its run time arguments, the client executes the series of system calls necessary to establish a connection with the server. This established connection is used to pass two messages to the server. The first message is the unique file name mentioned above. The second message is the data entered from the keyboard. Once the messages have been sent, the client closes its end of the socket connection and proceeds to create a new socket using the unique file name it created. It then blocks on the "accept" system call and waits for the server to connect. The server uses the file name passed in the first message to create a socket and connect with the client. This socket is used to pass the results of the query evaluation back to the client (Figure 9). This second socket raises the question of, if a socket is a two-way communication link, which it is, why create a new socket? In a strictly single user environment this second socket is unnecessary. However, in a multi-user situation it is necessary to free the first socket to receive connections from other clients. Since a multi-user environment is anticipated it seemed reasonable to design the system with this intent in mind. In this way each client will have a unique communication connection with the server.



A Client/Server Apparatus for the Volcano Query Evaluation System
Figure 9

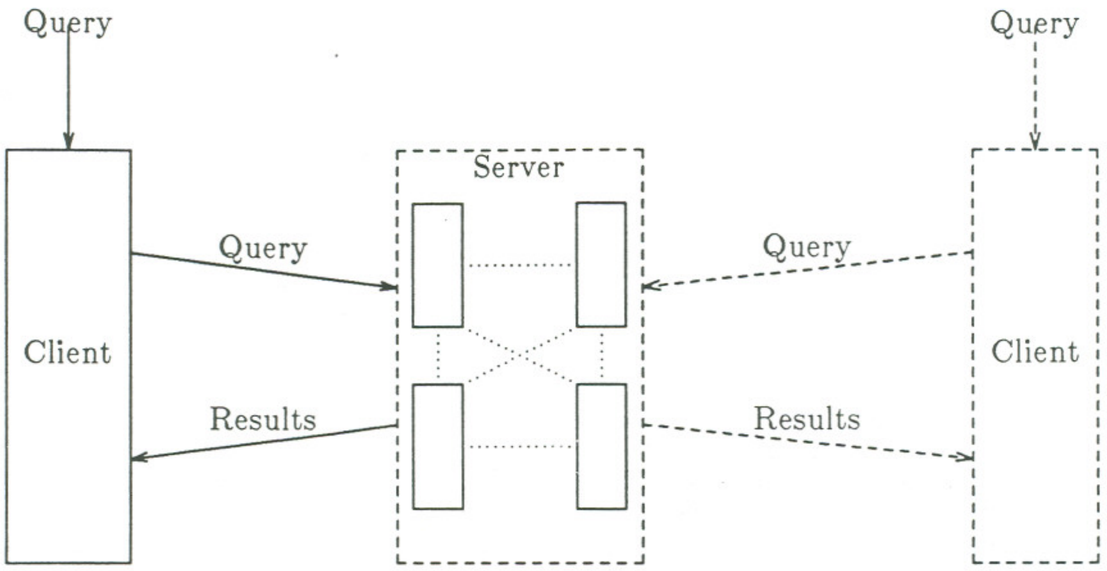
Once the output socket is connected, the server's standard output streams, stdout and stderr, are redirected through the socket. The compiled queries need no special structure to handle the output; the regular C function "printf" is sufficient. Requiring no special output structure is important since the print routines are supplied as support functions not as part of the Volcano library. The second message received from the client is separated into the query (a file name) and individual arguments, if any, to be dynamically loaded as described earlier, and executed. When the evaluation is completed, the next set of messages (a socket name and a query) is extracted from the input socket queue and the process is repeated.

When the session is complete, a "quit" entered from the keyboard will terminate the client. The server will remain in the background as an idle process. If "exit" is entered both the client and the server are terminated.

4.3. The Primed-Process Pool

The mechanisms described so far separate the query from the query evaluator. The query evaluator can persist for an indefinite amount of time and evaluate an arbitrary number of queries. The communication mechanisms for conveying the query from the keyboard or other input device to the query evaluator and the results back to the appropriate display device are provided. The next step is to extend the model to support parallel processing by adding a multi-server primed-process pool (Figure 10).

A primed-process pool is a group of identical server processes that are capable of working together on the same task. The term "primed" means



Client/Server with Multiple Server Processes
Figure 10

the processes exist prior to their actual need. The client/server paradigm discussed earlier is a special case of a primed-process pool where the pool is restricted to one process. We are interested in the more general case of an arbitrary number of processes.

For a process pool of an arbitrary number of processes to function together, we again have a need for interprocess communication. There are several levels of communication necessary. A process needs to be alerted that its services are required and supplied with the information necessary to perform them. Data needs to be passed between processes, and each process must be able to communicate the results to the appropriate output device.

An efficient system to pass the data between processes already exists in the Volcano system. This system, passing packets of record identifiers via shared-memory, is equally appropriate in the new system and need not be changed. This mechanism was discussed earlier in the Volcano section.

For the other communication requirements, all the communication methods discussed earlier are considered along with the Dynix pipe mechanism. Pipes were not considered earlier because they require the processes communicating to have a common origin, i.e., they are created using the "fork" system call. The client/server model discussed earlier does not have this property.

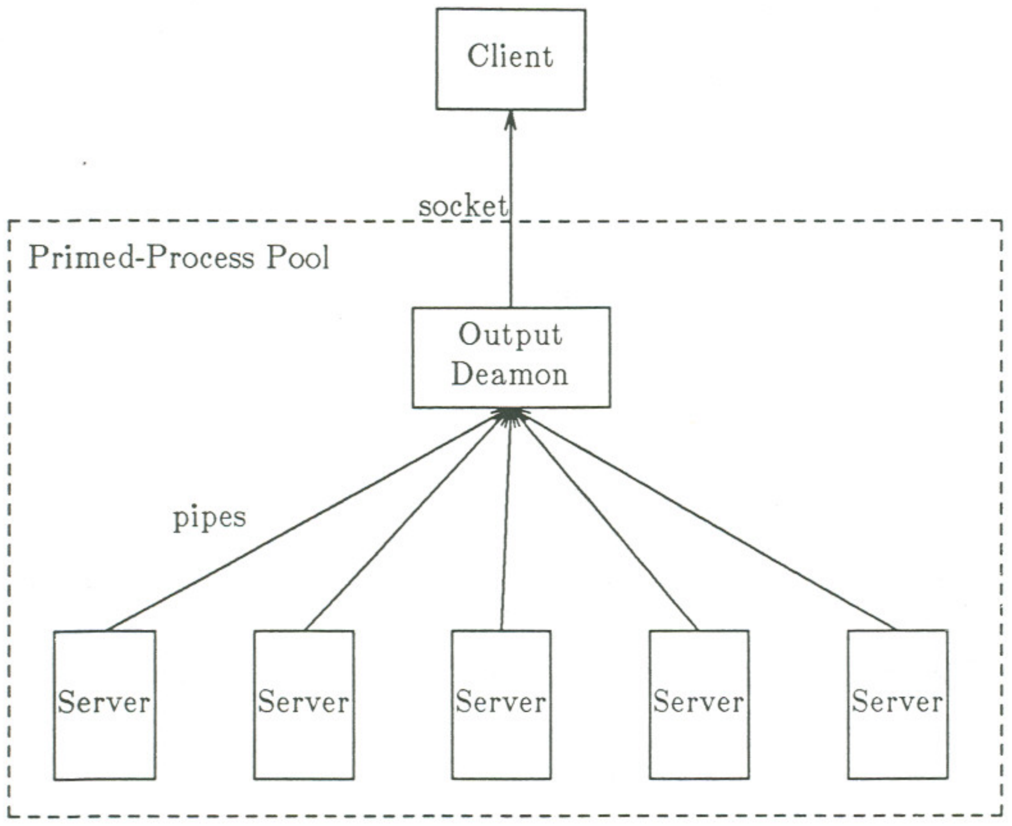
Pipes are implemented by the operating system using sockets. Two one-way communication channels are created, one for writing messages and one for reading. A pipe is created using the "pipe" system call. The system call returns two file descriptors. The first descriptor is for reading, the other is for writing. The "pipe" call is executed prior to forking more processes. Since each forked process is an exact copy of the original process, including the file descriptor table, it will inherit the pipe connection. The processes can then communicate via the pipe. The connection remains open until one of the processes involved closes the connection using the "close" system call. Once a pipe has been closed the connection cannot be re-established. If more than one process is forked, they will all be able to communicate over the same pipe. This establishes a many-to-many communication connection between all the forked processes. When a message is written using the pipe, it will be read by the first process that is able to read from the pipe. The messages cannot be directed to a particular process in the group.

In the primed-process pool there are three separate communication needs to address, alerting dormant processes, communicating between server processes and communicating between the client and the servers. Each need shall be addressed separately. To trigger a sleeping process within the process pool it is necessary to alert the process and send it the information necessary to begin execution. This information amounts to four integers: an identifier used by the exchange module to distinguish the master producer from slave producers; the address of the descriptor in shared-memory where the new process will begin execution (the state record); the bindings (which are currently not being used but will be used in future extensions to communicate environment variables between operators); and the address of the port used to pass data to the consumer.

The simplest way to accomplish this communication is by using the pipe mechanism described earlier. If all the slave processes are blocked on a read from the pipe, then simply writing the message will alert a process that its services are needed. A separate message is written for each process required. The fact that the messages cannot be directed to a particular process is not a concern since all the slaves are identical. With either the memory-mapping or a shared-memory scheme the alert signal and the information transfer would be separate actions.

The communication between the client and multiple servers is more complex. Dynix doesn't support a many-to-one communication connection between processes that do not have a common ancestor, as in the case of the servers and client. Pipes can't extend beyond their ancestral group and sockets can only connect two processes at any one time. Memory-mapping is an alternative that was explored. A file is mapped into the space of a client. The servers write into the file and the client simply writes the data directly from its address space to the output device. This method has the attractive feature that the information is not written and read multiple times. It suffers, however, from two major shortcomings. First, having multiple processes writing into a common space requires synchronization. Since the output is generated by support functions supplied with the query, it is difficult to synchronize. Even if it were possible, it is not desirable. One of the major attributes of Volcano is its ability to do parallel processing without a great amount of synchronization. Adding another layer of synchronization would cause unacceptable degradation in performance. The second problem with this design is that the client needs to check continually to see if there was information in its buffer to be written out. This check would cause the client process to be in a busy wait or to be activated at some fixed interval of time. The client would be using CPU cycles regardless of whether there was any useful work for it to do. This situation is not tolerable in a high performance system.

In the absence of any direct many-to-one communication facilities we opted for an indirect method using pipes and a socket (Figure 11). The output generated by the servers is redirected through a pipe. One designated process in the group blocks on a read from the pipe. This process is also connected to the client via a socket. When data is read from the pipe it is then written to the socket. This scheme suffers from the obvious drawback that the data is read and written several times before it reaches its destination. It also requires an extra process. This scheme does have several advantages. The design does not require any special print routine in support functions, nor does it require the client to be activated arbi-



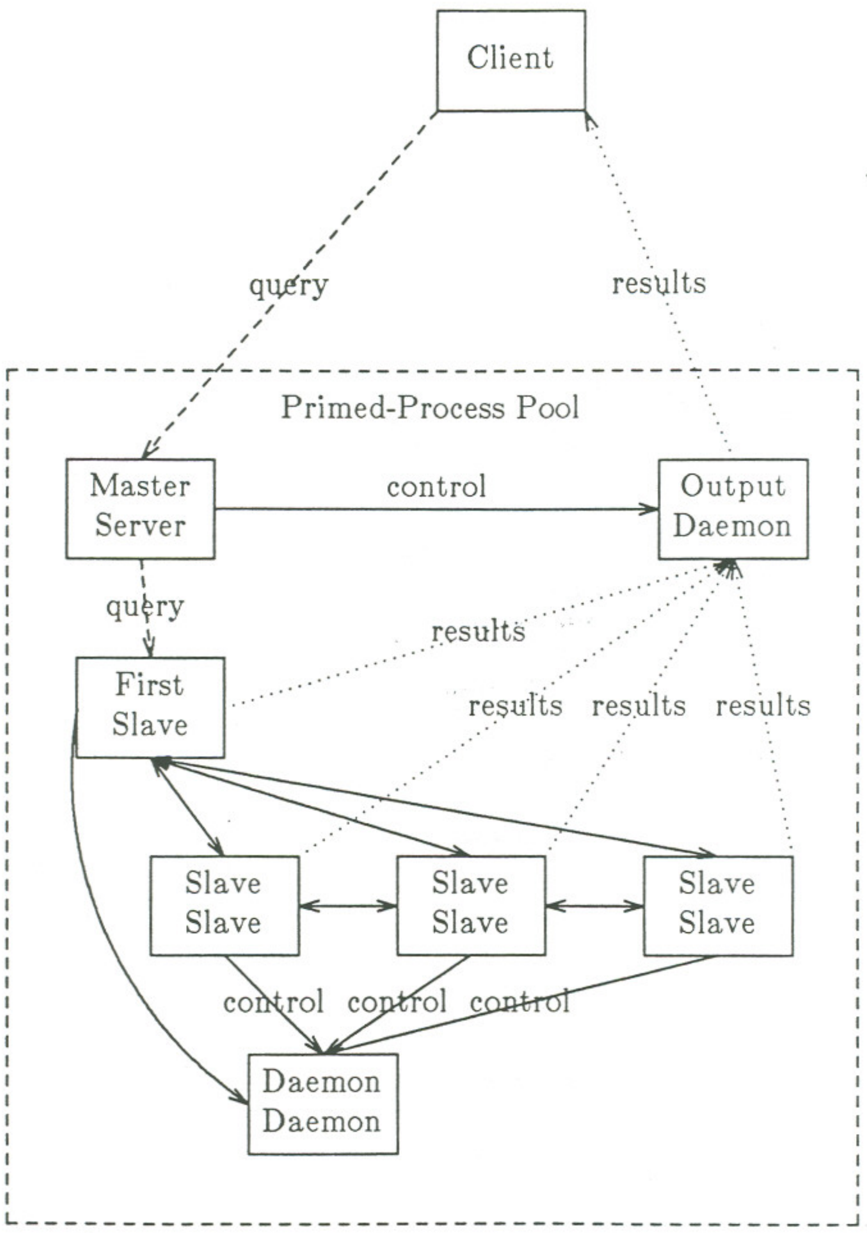
The Output Communication Mechanism for the Primed Process Pool
Figure 11

trarily to check for data. This option seems to be the cleanest and most efficient of those available.

The primed-process pool is implemented with five process types: the master server, the output daemon, the first slave, the slave slave, and the daemon daemon (Figure 12). In the current single user system there is one instance of the master server, the first slave, the output daemon and the daemon daemon. There can be an arbitrary number of slave slave processes. The first four process types (master server, output daemon, first slave and slave slave) are discussed here, the discussion of the daemon daemon is deferred until the next section.

When the primed-process pool is first executed a single process is started. This process allocates space in shared-memory for the Volcano buffer and establishes two pipes for communication between the processes in the server pool. One pipe is to communicate messages between server processes and one is for communicating results to the appointed output daemon, i.e., information that is destined for the client. The process then forks slaves. The number of slaves forked is a command line argument. If no argument is given, there is a default, which is a compile time constant. The original process assumes the role of the master server.

The master server creates a socket with a name that is known to the clients, it then blocks on the "accept" system call waiting for a client to request service. When a request is received, the master server activates the output daemon (the appointed communication process) to create and connect the return socket to the client. The master server then activates the first slave, passing it the query and any parameters received from the client. The master server can now wait for another request from a client.



Communication Links for the Primed-Process Pool
Figure 12

The output daemon handles the output duties of the process pool. After connecting the output socket to the client, the output daemon blocks on a read from the output pipe. To help reduce the overhead of the multiple reads and writes, the output from the pipe is buffered. The results are only written to the client when the buffer is full or when all the outputs have been received from the slave processes. Buffering reduces the number of read and write system calls necessary to handle the output as well as reducing the frequency with which the client is activated. The size of the buffer is a compile time constant. It is currently 1024 bytes.

The first slave is the process that actually begins executing the query. It calls the procedure that executes the dynamic load. The query is loaded and executed as described earlier. If more processes are needed to evaluate the query, they are activated by the exchange module using the *start_new_process* routine (to be described later). When the first slave has completed its part of the evaluation, it waits for the other slaves, if any, to complete. When all the slaves have completed, the first slave alerts the output daemon that the evaluation is complete so the output daemon can flush the output buffer. The first slave then executes a routine to flush the Volcano buffer and waits for a new query to evaluate.

A slave slave is activated when more than one process is required to evaluate the query. It is passed a message consisting of the integer values discussed earlier. These integers are used as arguments to the execute_producer function which is part of the exchange module. The modifications that were necessary to the exchange module will be discussed shortly. A slave slave process may be activated by the first slave or by another slave slave. The activation is always done by a call to the start_new_process from the open exchange function. A global active process counter is incremented for each new process started. When the execute_producers function completes, the slave slave process decrements the active process count and waits for another request for service.

In the current implementation, the master server and the output daemon are the same process. In a single-user environment it wasn't necessary to use two processes. This arrangement also acts as a safeguard to prevent multiple queries from being introduced into the system. The introduction of multiple queries is possible since the system does support multiple clients. When the environment is extended to support multiple users and the proper concurrency controls are in place, the output daemon can easily be moved to a separate process. All the code necessary is encapsulated in one routine. With multiple users it will be necessary for each user to have their own output daemon process. This extension will require a separate output pipe for each output daemon. Separate pipes are necessary because messages sent through pipes cannot be directed to a particular receiver. If only one output pipe were used, the results might not be transmitted to the correct user.

The only process in the process pool that is restricted to a particular task is the master server; all other processes may assume any role. A process may be the first slave for one query and a slave slave for the next. This flexibility allows all inter-server communication to be done on one pipe, which is important in an environment where the number of open file descriptors is limited. Keeping the processes as general as possible will also make extending the system to a multi-user system easier.

To achieve parallel processing in the previous version of Volcano, the open_exchange function forked a master producer which immediately executed the execute_producers routine. The execute_producers routine would create a port to pass records to the consumer(s), fork the required number of slave producers, and continue with the parallel execution of the query.

In the current implementation the open_exchange function creates the port and then calls the start_new_process procedure. This procedure maintains a count of the number of active processes. If there is not a sufficient number of processes available, the daemon daemon is invoked. The start_new_process procedure then sends out the appropriate number of messages on the inter-server pipe to start the appropriate number of slave slaves. The slave slaves then execute the execute_producers routine. This procedure now receives the port address as an argument, and the code to execute the fork has been removed. The only other change needed was to re-initialize some global variables after execution completes.

As mentioned earlier, the query is loaded in shared space. The state records, which are part of the data structures included with the query, are written to by each process. Some of the information stored in the state record is unique to the individual process, it is therefore necessary to copy the state records for each operator into private space. This copying is done by the open function of each operator. Any further references to that state record are made to the private copy rather than the original version in shared space.

4.4. The Dynamic Primed Process Pool

Since the servers are independent of the queries it is not possible to know a priori how many processes are necessary. If a query requests more processes than are available, the query will stop, waiting for a nonexistent process to complete. This situation could be prevented by starting enough processes at the time the server is booted to accommodate the most complex query. Starting a sufficient number of processes requires a great deal of omniscience on the part of the user, or spawning an excessive number of processes at boot time. Neither situation is desirable. To resolve this problem in a more satisfactory way we have implemented a daemon daemon.

The purpose of the daemon daemon process is to fork a specified number of processes on request. When the number of processes available for use drops below a given threshold, the daemon daemon is activated. The daemon forks the requested number of processes then goes back to sleep. The requested number is either the number required to maintain the threshold or a predetermined minimum, whichever is greater. Both the threshold value and the minimum are compile-time constants. When the start_new_process function is called, it compares the number of active processes plus the number of requested processes against the total number of processes. If the number of needed processes is within the threshold of the total number of processes, then the daemon daemon is activated. If the requested number of processes is greater than the minimum, then the requested number of processes are forked, otherwise the minimum number are forked. i.e. The threshold is 5, the total number of processes is 20, the minimum number to start is 6 and currently 14 processes are active. A request for 4 new processes is made. 20 - (14 + 4) is less than the threshold, 5 so the daemon daemon is activated. The minimum,6 is greater than the number of processes to be activated, 4 so 6 new processes are created. There are now a total of 26 processes, 18 are active. With this scheme there should always be at least the threshold number of processes available for service.

Variables containing a count of the total number of processes and the number of processes in use are stored in shared memory. To keep the numbers correct a lock is maintained when these values are updated. In the worst case the lock is held for one comparison and two updates. When a process has completed execution one decrement is necessary to the "processes in use" variable. Since the lock is held for only short periods, performance should not be affected.

The daemon daemon is implemented as part of a slave server, rather than a special process. The actual code executed by a slave server is controlled by a flag in the message that activates the process. In this way, when more processes are necessary, the first available slave server will become the daemon daemon. After all the processes requested have been forked, the daemon daemon will again be available as a slave server.

There is currently no mechanism to reduce the number of processes. If it is found to be necessary, it can easily be added as part of the cleanup procedure after a query is processed or after a session is completed.

4.5. The Distributed Primed-Process Pool

Volcano is designed as a tool to explore database design policies, including different machine configurations. One configuration that we wish to explore is a network of shared-memory machines. The current implementation of Volcano is incapable of functioning beyond the boundaries of a single shared-memory machine. The data distribution mechanisms used by the system are dependent on a shared-memory architecture. The mechanisms necessary to extend the system across machine boundaries are thought to be similar to those developed for the primed-process pool. Therefore, it is an appropriate time to explore this extension.

The problem of extending Volcano across machine boundaries can be broken into four major sections, collecting the data to be sent to a remote machine, re-introducing data received into the Volcano system, the communication mechanism necessary to transmit the data between machines and a communication mechanism to transmit the results to the correct user on the correct machine.

Within a shared-memory machine records can be passed from one process to another simply by passing a pointer to their location in the shared buffer. Passing pointers to memory locations obviously will not work with multiple machines. The actual records need to be passed. A mechanism to accomplish this needs to be developed. A mechanism is also needed to insert the records received from a remote machine into the Volcano buffer space of the requesting machine. These records must be inserted in such a way that they can be processed by the current Volcano system as if they were generated locally.

The basic tools necessary to collect records already exist within the Volcano buffer management system. Space must be allocated within the buffer to create a packet of records to be transmitted to a remote machine. This space allocation can be done by creating a file on a *virtual device*. A virtual device is a data structure within the Volcano buffer. It has the same basic characteristics as a real device or file with one exception; it does not exist outside of the buffer. Once all the records in a file on the device are unfixed the file disappears. Files are divided into units called *clusters*. The cluster size of this file will be the size of a data message. As the records required by the requesting machine are found or created, they are appended to the file. When a cluster is full, the records are shipped to the appropriate machine. The cluster can then be unfixed in the buffer and a new cluster begun.

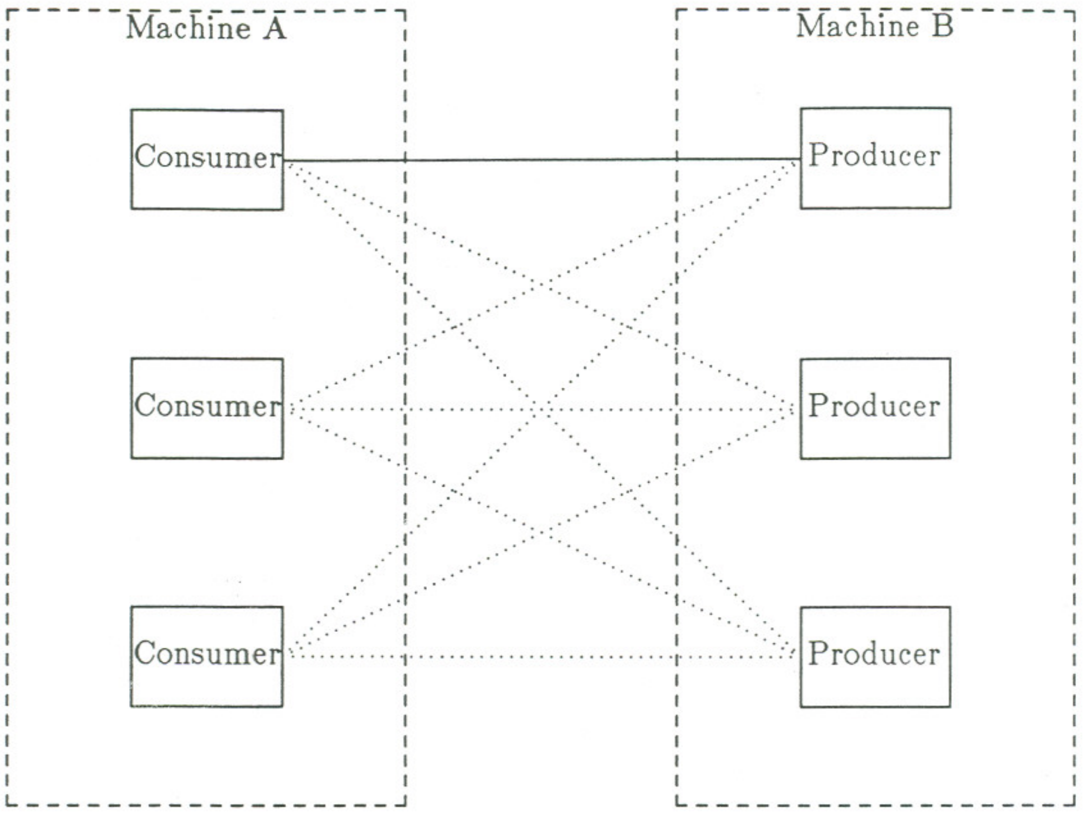
When the records are received by the other machine, they are read into a new cluster in a file on a virtual device. If these records were created on the remote machine, i.e., the product of a join, they will have no meaningful record identifiers associated with them. Therefore, it is necessary to treat these records differently than the records generated and processed within the same machine. The records in the cluster must be scanned (as the file_scan operator would scan a file) and record identifiers assigned. The records can then be introduced back into the usual Volcano stream, passing the record identifier (the NEXT_RECORD structure) on to the next operator.

To extend Volcano across machine boundaries, several levels of communication are required. The initial message sent to a remote machine will consist of the compiled query. As a separate message, the point in the query tree at which the remote machine is to begin processing must be supplied. This, "startup message", is considered as a separate message because the query need only be transmitted once while the machine may be requested to perform several operations on the same query starting at different locations in the tree. These messages can originate from any process on the requesting machine and are sent to the process pool on the remote machine. The query is loaded into shared space and can be accessed by any process in the pool. The records being processed must also be sent between machines. They will be sent from the process generating the records on one machine to a particular accepting process or group of processes on the second machine.

The only inter-machine communication mechanism available in the Dynix environment is the socket paradigm discussed earlier. Establishing the socket names for inter-machine communication is slightly more complicated than intra-machine communication. It involves the machine host name and a physical port address. The limitations on the socket paradigm are the same: only two processes can be connected by a socket at any one time, and it takes several system calls to establish a socket. With these limitations in mind several options are considered.

The most direct approach is to allow the process requesting the remote machine to establish a connection directly to a server process on the remote machine. .This approach entails creating a socket connection to a known socket address on the remote machine, and sending the necessary information. Ideally, considering the overhead involved in establishing a socket connection, we would like to maintain this connection between machines. Unfortunately, maintaining this connection is not practical for three reasons. First, the process that sent the message may not be the one that will process the information that is returned. This information may be handled by several processes, e.g., multiple consumers. Second, the connection is made to a known address on the remote machine. It is not possible for machine B to maintain a socket for every process on machine A waiting for a potential connection. Machine B cannot know at boot time how many processes machine A has or may have in the future, and machine B can only maintain a limited number of socket addresses for use by other machines. Third, the process on the remote machine that received the message may not be the process that will generate the data requested. There may be multiple producer processes on the remote machine generating data for the consumer(s) on the other machine(s) (Figure 13).

A second alternative is to use a single process to do the communication between machines. This design is similar to that used between the clients and the server pool on a single machine. This communication daemon would establish a socket connection with a remote machine. When a server process on machine A had a message to send to machine B, machine A would transmit the message to its communication daemon which in turn would send the message to machine B. This scheme has the advantage that the inter-machine connection need only be established once. Each



——— Established Socket Connection

·········· Other Required Connections

Process to Process Communication
Figure 13

machine would have a single channel to each auxiliary machine by which to transmit information. Having a single persistent channel between machines eliminates the overhead of creating new socket connections, for each message. This design does, however, require each message be read and written an additional time for each message sent and possibly for each message received (on the receiving end an additional read and write may be necessary depending on how much work the receiving process does itself and how much it passes on to other processes). Measured by the number of system calls required, this design will still be cheaper than opening and closing sockets. A socket requires two system calls to request a connect, one to accept the connection, a "read" and a "write" call, and a "close" call on each end to terminate the connection. There is a total of seven system calls per message. With the central communication daemon there would be at most six per message, two writes and one read to send, two reads and one write to receive. Having centralized communication could prove to be a performance problem. A bottleneck may occur when the records are being passed between machines. The transmission of the query and the message supplying the starting location in the query tree would not be a problem. The query is transmitted only once, as the initial message, and the starting location messages are short and relatively infrequent. Sending all the data records through one process may prove to be a problem. The actual costs involved need to be explored further.

A third alternative is a combination of the first two. The query and the starting point messages could be transmitted using the centralized communication daemon. The data records would be transmitted directly from the process on machine B generating them to the process on machine A that will process them. A unique socket connection between a consumer on machine A and the producer on machine B could be established by including in the starting point message a socket name that the consumer is waiting on. With a perfunctory examination, this alternative seems viable. However, a closer examination reveals two fatal flaws. The one-to-one link structure of a socket restricts the consumer-producer relationship to be one-to-one. In most situations this relation is actually one-to-many or many-to-many. The second flaw requires the consumer process on machine A to process records received from producers on machine A as well as deal with the communication link to machine B. One process cannot block on a read from a socket and process records concurrently. To execute both functions concurrently two separate processes are required, which suggests the second alternative may be the most workable.

So far the discussion has focused on the sending end of the connection. We now address the receiving end. The machine receiving the message has two methods with which it can handle messages from multiple machines. There can be one process that receives all the messages from all the machines, or there can be a receiving process associated with each remote machine. The single process scheme requires the sockets to be multiplexed. Multiplexing is accomplished with the "select" system call[7].

When using the "select" system call, the sockets to be multiplexed are created, bound to a name and marked ready to accept a connection. The socket, bind and listen system calls are executed. A bitmap is created using the file descriptors as indices, indicating the file descriptors to wait on. The process will then block on the "select" call. When a connection is pending on a multiplexed socket, the select call executes, zeroing all bits in the bitmap except the one associated with the pending connection. The indicated connection can be accepted and the message transmitted in the usual fashion. If the "select" call is to be used again to receive new messages, the socket must be closed and the bitmap reset. The performance problems associated with repeatedly connecting and reconnecting sockets were discussed earlier.

The second alternative has a process whose sole purpose is to process messages from a specific machine. It would establish a connection with a remote machine and that connection could exist as long as the connection was deemed useful. This design has some clear advantages over the previous design. The socket reconnection overhead is eliminated. This design would be very easy to extend as new machines were added to the system. A slave server could easily be converted into a communication daemon for the new machine. Likewise, if a machine was removed from the system the communication daemon could be terminated or converted back to a slave server. To add or delete machines in the first scheme would require the server pool to be recompiled.

When a remote machine is needed to assist in evaluating a query, the first information it will need is the query to evaluate. The query can be transmitted in various forms, as a source file, an object file or an as executable code. One assumption we make about our distributed system is that all the machines are identical (at least to the point where code could be compiled on one machine and executed on the other). Without this assumption the source file have to be sent.

The form of the query that is the most convenient to transmit in the current test environment is not the best for the system in general and vice versa. In the more general case the object file would be sent to the remote machine via a socket connection. The remote machine would allocate the space necessary and execute a dynamic load. Any subsequent starting location messages received would be relative to the head of the dynamically loaded file. With this design there is effectively no restrictions on the size of the query or on the location on the remote machine it is placed.

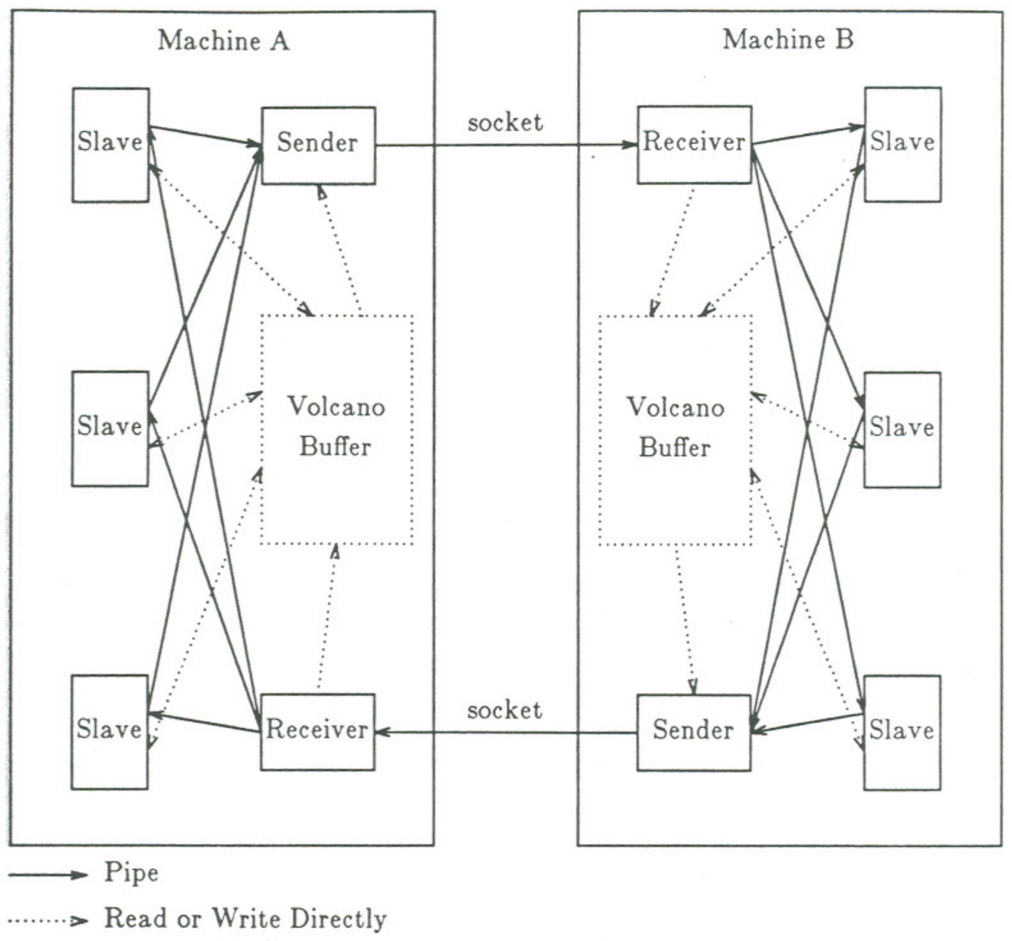
In the current test environment it is more convenient to send the executable code, the actual data and text segments of the query as it exists in the executing program. In the test queries some of the parameters are set at run time, such as the number of processors requested. If only the object file is sent, these parameters will never get set. When the query is passed between machines in the form of executable code, it is necessary that the code be loaded into the same relative address on the remote machine as it is on the original machine. If the code is not loaded into the same relative location on the new machine any references to the code will be incorrect. To guarantee the code is loaded into the correct location, it is necessary to statically allocate the space into which the query is loaded. Statically allocating space strictly limits the size of the query. The simplest way to avoid this restriction is to rewrite the test queries, not using command line arguments. The command line arguments have proven to be convenient for testing purposes. It is, therefore, this author's recommendation that the second design (transmitting the executable code) be implemented until the testing phase of the project has been completed or until there is a more convenient means to implement queries. The conversion to the more general method should be straightforward.

A separate communication link is used to return the results to the user. Part of the mechanism already exists, the output daemon process. The remote machine's output daemon can write the output on a socket to the requesting machine rather than to a client process (with a single-user paradigm there would be no client process on the remote machine). The receiving machine could use one of the schemes discussed earlier for receiving the results. It is necessary to separate the communication links for the results from that of the records to be processed because they do not necessarily go to the same machine. As an example, machine A starts a query, then calls machine B for help. Machine B in turn calls machine C for help. The records that machine C is producing will be returned to machine B for possibly more processing. However, if any output bound for the user is generated, it can go directly from machine C to machine A for display.

Using the previous discussion as a basis we now discuss an implementation design. Each machine has two communication processes for each remote machine. One is used for sending information and one for receiving information. This scheme eliminates the need to open, close and reopen sockets during query processing. This scheme will increase the number of processes contending for a limited number of processors on each machine. This contention may cause extra context switches, which are also expensive. The guaranteed expense of opening and closing sockets would probably be a much greater detriment to performance than the possible processor contention.

The sending process is connected to the server processes by a pipe. The pipes are established at the time the process pool is booted. The number of pipes established is the limiting factor in the number of machines that can be connected in one system. If more machines are needed, the process pool needs to be rebooted with more pipes established. When a server process on machine A (Figure 14) has information to send to a remote machine B, it sends the information through the pipe associated with the machine B to the sender process associated with machine B. The sender process then forwards the message to machine B via a socket (Figure 14).

The receiving process on machine B responds in different ways depending on the type of message sent. Each message sent contains a header that can be examined to determine the type and size of the message. If the message contains a query, it can be loaded dynamically as done on the original machine (or if the current test queries are being used, simply read into the preallocated space). The output daemon needs to be started and told where the results should be sent. In the case of a starting location message, the receiver process adjusts the starting location to correspond to the correct address on its machine. Along with the starting location, this message should include the number of processes that are being requested



Intermachine Communication
Figure 14

and some means of identifying the process(es) on the sending machine that will process the returning data. The requested processes can then be started using the start_new_process routine. This routine will be modified so it can alert the processes it starts that the records they produce need to be packaged and sent to the appropriate consumer(s) on the remote machine.

If the message received includes data, the receiving process will read the data into a cluster on a virtual device. The consumer process(es) waiting for the records will be alerted that data is ready for processing. The exact mechanism to alert the consumer and transmit the cluster location needs further investigation.

Many of the designs discussed, including several that were not recommended, were tested in a partial implementation of the distributed system. Due to a limitation of time and machine resources (particularly the number of open files allowed under the current system configuration), the full design as described above was not implemented. Our trial implementation attempted to use a direct "process to remote machine" approach. The process requesting work from the remote machine creates a socket, sends the required message, closes the socket, then sleeps until data is returned. All messages are received by the master server. The messages are handled as described in the above design. When the records are returned to the requesting machine, the dormant process is reactivated and the records received are introduced into the normal Volcano processing stream. We hoped that a method for multiplexing the connection could be developed that did not require continually reconnecting the sockets. When this method was not forthcoming, this implementation was discontinued.

The original implementation of the primed-process pool required most of the file descriptors available. Subsequent revisions have freed several descriptors. It should now be possible to attempt, at least a limited implementation of the recommended design.

The design of the distributed primed-process system is heavily dependent on facilities provided by the operating system. The most important facilities an operating system could provide for this type of project is a fast, reliable interprocess communication system. This system should include a direct many-to-one or many-to-many communication facility capable of communicating across machine boundaries. This project would also benefit from the ability to address messages to a particular process or processes.

CHAPTER 5
PERFORMANCE TESTING

The main goal of this project was to reduce the overhead caused by creating processes using the "fork" system call. To determine if we accomplished this goal, tests are designed to measure the cost of this overhead and not the cost of evaluating a query, which is largely unaffected by the primed-process mechanisms. The queries used in the tests consists of two types of operators, the *exchange operator* and the *create_file* operator. The create_file operator creates a file with no records in it. This operator is included to supply the exchange operator with a child node in the query tree. The system is not designed to have an exchange module as a leaf of a tree. This query is effectively doing no work, so the cost of the execution is determined entirely by the creation or activation of new processes by the exchange module.

Three different query trees were used for the tests. The first tree included one exchange module, the second, three and the third, five. The first query tree was run with the degree of parallelism varying from one to eight. The second and third query trees were run with one, four and eight processes per exchange. The cost of various operations were measured as well as the total time required for the query to complete. The total time was taken from the time the query began executing to the time a second query could be entered. This time included the initialization and cleanup costs involved in each system. In the primed-process system individual costs were measured for the incremental load, the connection of a socket, the cost of reformating the devices after the query completed and the cost of the actual execution of the query. For the system using forking, the query execution was again measured as well as the cost of mounting and unmounting the devices. The times were measured as elapsed times using the system microsecond clock. Each test was run ten times, five to collect individual times and five to collect overall time. The tests were run in two sets so the cost of collecting the individual times were not included in the overall times.

The tests were designed to be run on the Oregon Graduate Center's eight-processor Sequent Symmetry. The tests were actually run on a fourteen-processor Symmetry that the Sequent Corporation generously allowed us to use. This machine is used specifically for experimentation and benchmarking.[1] We had exclusive use of the machine for the duration of the tests. The memory allocated by the Volcano memory management system was relatively small, five megabytes. Attempts were made to run the tests with a much larger buffer, fifty megabytes. The test would not run as designed. Some problems with the operating system had been discovered earlier and these problems were presumed to be the cause of the restriction on available space for the buffer.[2] We will, hopefully, be able to rerun these tests in the future. We suspect that the larger buffer, requiring a larger page table, has an effect on the cost of forking processes.

5.1. Results

A summary of the results of queries 1, 2 and 3 are shown in Tables 1, 2 and 3 respectively. The times shown are an average of the middle three results of each set of five tests. In some cases one of a set of five test results were two to three times greater or smaller than the other four results. Values that were very high or low were considered to be measurement errors or influenced by the operating system and thus ignored. The

[1]The machine was equipped with experimental processors that deliver about 5 MIPS per CPU.

[2]The machine was also being used with experimental versions of the Dynix operating system. We suspect that an experimental version was running at the time. The operating system modifications affected file system operations, not process operations, so we are confident that our measurement are representative of the production operating system.

times are expressed in seconds. The "exchange cost" reflects the cost of forking or acquiring processes. The "total costs" includes all the costs from the time the query is first read to the time the results are printed on the screen. Tables 4 and 5 shows the cost of various components of the total costs.

Cost* with 1 Exchange Module				
number of processes	exchange cost		total costs	
	forked	primed	forked	primed
1	.065	.021	.802	.930
2	.199	.019	.986	.926
3	.235	.018	1.034	.931
4	.279	.019	1.089	.922
5	.311	.020	1.135	.910
6	.358	.022	1.191	.906
7	.362	.024	1.203	.900
8	.394	.027	1.275	.922

* cost is elapsed time in seconds
Table 1

Cost* with 3 Exchange Modules				
number of processes	exchange cost		total costs	
	forked	primed	forked	primed
1	.140	.043	.852	.950
4	.465	.051	1.284	.960
8	.723	.093	1.685	1.044

* cost is elapsed time in seconds
Table 2

Cost* with 5 Exchange Modules				
number of processes	exchange cost		total costs	
	forked	primed	forked	primed
1	.215	.055	.937	.997
4	.659	.081	1.497	1.060
8	1.110	.1562	2.026	1.094

* cost is elapsed time in seconds

Table 3

Figure 15 shows a comparison of the process acquisition with the primed-process system versus forking processes in the original system with one exchange module. Using primed processes clearly provides better performance than process forking. Figures 16 and 17 using three and five exchange modules, respectively, also illustrate the same improvement.

Figures 15, 16 and 17 show how the cost of forking processes is directly dependent on the number of processes forked. The cost per process decreases with the number of processes forked because of the propagation scheme used. The primed-process system also increases with the number of processes but at a much slower rate. As the number of processes required increases, the benefits gained by using primed processes also increases.

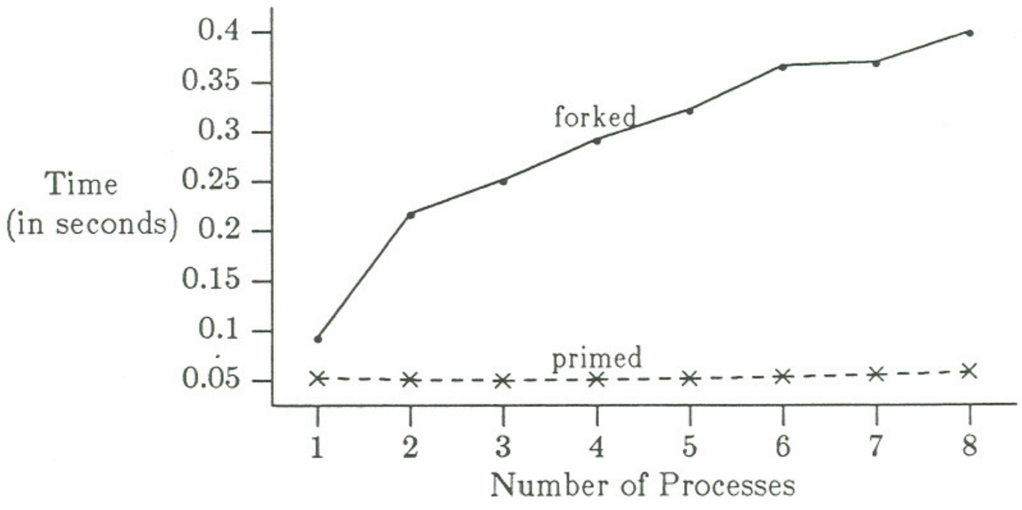
Figures 18, 19 and 20 compare the total costs of each of the query trees tested. The total cost of the primed-process system is dominated by

Fixed Costs* for the Primed Process System	
Action	Cost
socket connection	.008
incremental load	.643
format device	.105

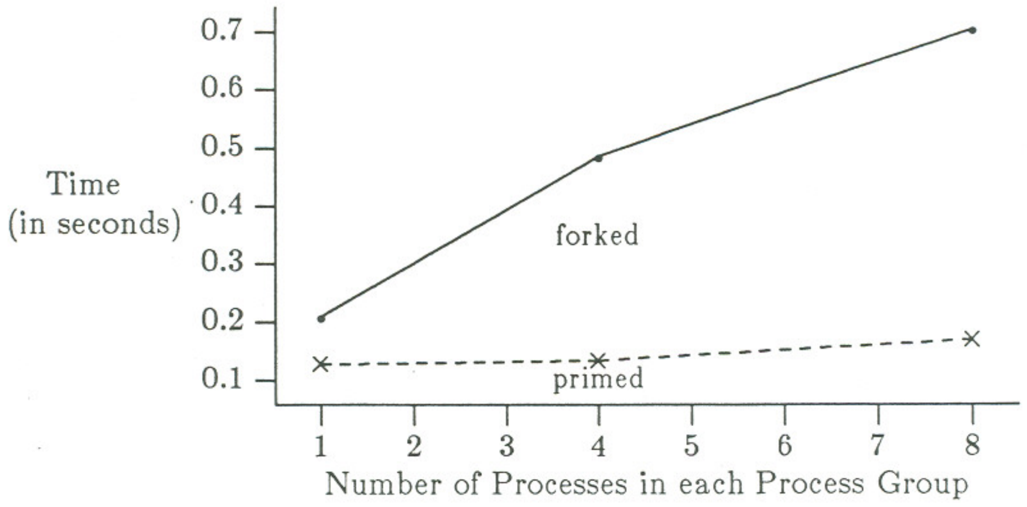
* cost is elapsed time in seconds
Table 4

Fixed Costs* for the Forking System	
Action	Cost
mount device	.716
dismount device	.095

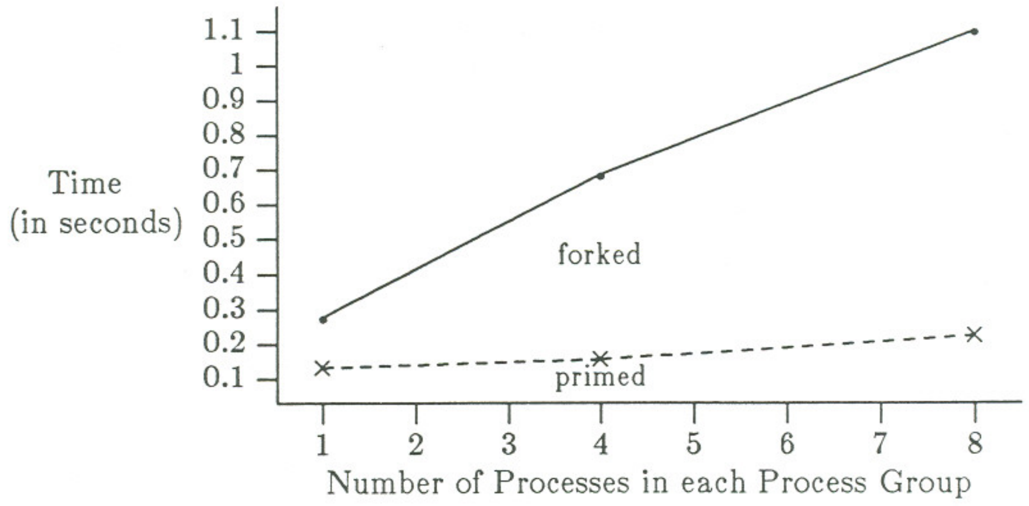
* cost is elapsed time in seconds
Table 5



Overhead of Query with 1 Exchange module
Figure 15

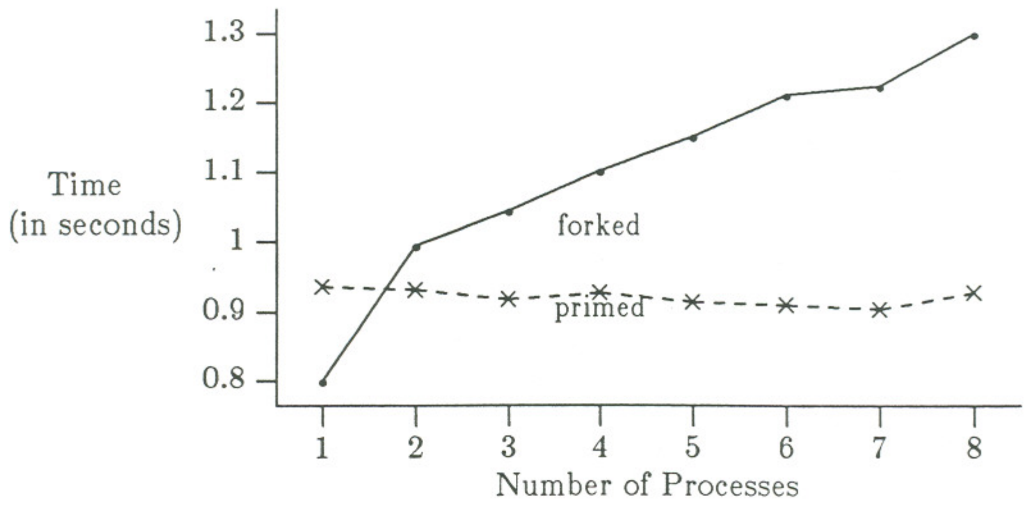


Overhead of Query with 3 Exchange modules
Figure 16

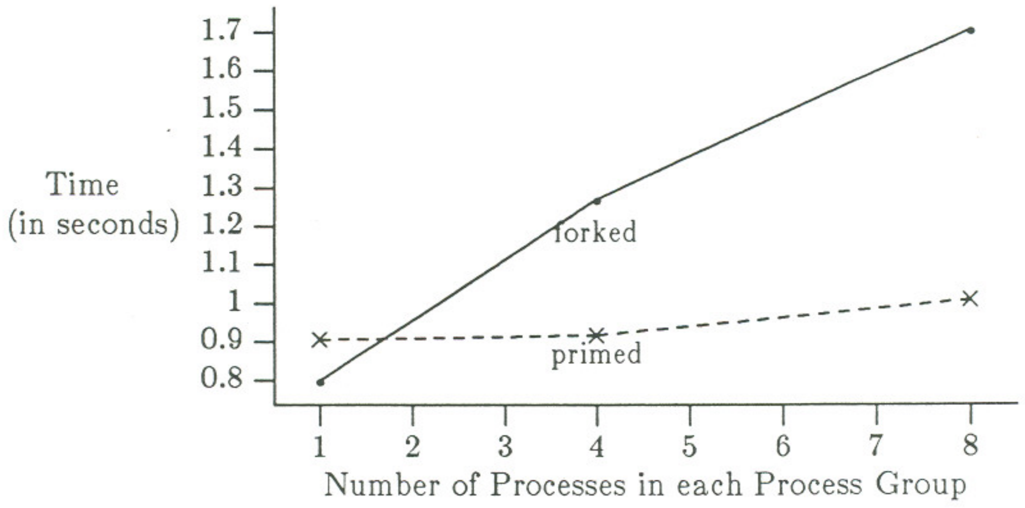


Overhead of Query with 5 Exchange modules
Figure 17

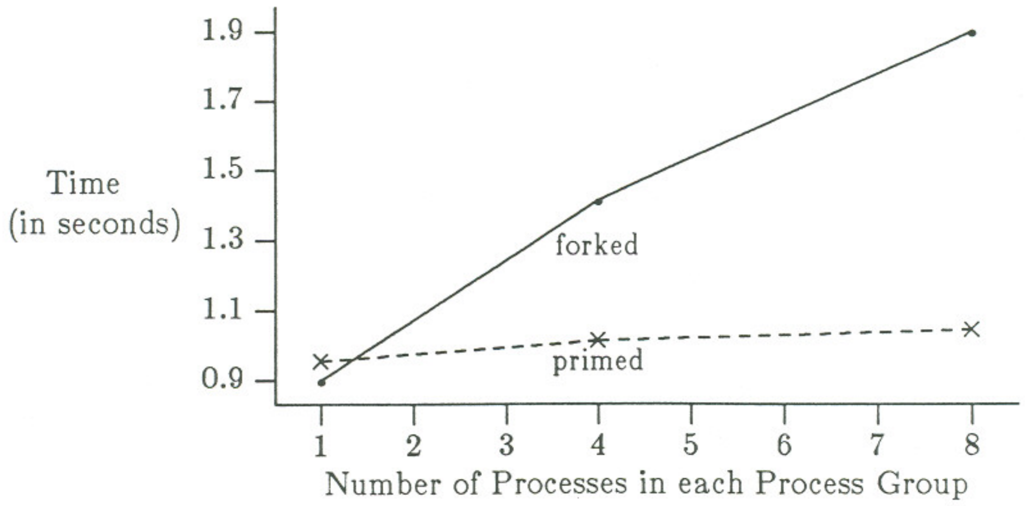
the cost of the incremental load. Even with five exchange modules using eight processes each, the incremental load is over half the total cost (Table 3). The indirect method used to do an incremental load will account for some of the cost. The link editor is accessed by a Dynix command, not a system call. To invoke the link editor the system call "system" was used. This system call creates a new shell process (forks a process) to execute the command. The link editor must also read the object file being loaded and the symbol table of the executing process. This work requires two reads from a device. The total cost in the forking system includes the cost of allocating and deallocating space in memory as well as the cost of executing the query. When the total costs are considered, the primed-process system still reduces the overhead when more than two processes are required.



Total Overhead of Query with 1 Exchange module
Figure 18



Total Overhead of Query with 3 Exchange modules
Figure 19



Total Overhead of Query with 5 Exchange modules
Figure 20

CHAPTER 6

SUMMARY AND CONCLUSION

The performance tests have clearly demonstrated that the primed-process system requires less overhead than the previous system. What is more significant than the actual numbers is the rate at which the costs increase. The overhead for simple queries with little parallelism is small in both systems. With the primed system this overhead is almost constant, increasing very little with the complexity of the query. The overhead of the forked system increases rapidly with the complexity of the query. This behavior suggests the performance gains of the primed-process system will be quite significant for complex queries.

We anticipated if very large main memories are available they will become an important part of high-performance database systems. If, as suspected, the cost of forking increases with the amount of memory a process has allocated, the cost of forking processes will become very significant even for simple queries. Earlier tests measured costs up to 700 milliseconds to fork a process with a 70 megabyte buffer. Starting processes in the primed-process system does not involve copying process structures and page tables, as does the forking system. This copying adds significantly to the cost of forking processes. The increase in memory allocation should, therefore, have no effect on the primed-process system. Thus, the performance gain will be even more significant.

We have implemented a dynamic primed-process pool for the Volcano query evaluation system. The evaluation system can exist independently of the queries it is to evaluate. It can evaluate multiple queries without being recompiled. By changing the "fork on demand" paradigm to a system of pre-existing processes we have significantly reduced the overhead of starting a new process. This change has improved the overall performance of the system. Many of the mechanisms necessary to extend Volcano to a distributed system were explored. Many of those used to implement the primed-process pool can be extended to work in a distributed system. The basic elements to extend the system to a multi-user system are also now available.

The next phase of the project should be to implement the distributed primed-process pool. While it should be possible to have a limited implementation with the present equipment and system configuration, it may be necessary to do the implementation in a different environment more suitably configured for the project.

REFERENCES

1. D.J. DeWitt, R.H. Gerber, G. Graefe, M.L. Heytens, K.B. Kumar, and M. Muralikrishna, "GAMMA - A High Performance Dataflow Database Machine," *Proceedings of the Conference on Very Large Data Bases*, pp. 228-237 (August 1986).

2. W. Alexander and G. Copeland, "Process and Dataflow Control in Distributed Data-Intensive Systems," *Proceedings of the ACM SIGMOD Conference*, pp. 90-98 (June 1988).

3. S. Fushimi, M. Kitsuregawa, and H. Tanaka, "An Overview of The System Software of A Parallel Relational Database Machine GRACE," *Proceeding of the Conference on Very Large Data Bases*, pp. 209-219 (August 1986).

4. M. Stonebraker, R. Katz, D. Patterson, and J. Ousterhout, "The Design of XPRS," *Proceedings of the Conference on Very Large Databases*, pp. 318-330 (August 1988).

5. D.J. DeWitt, M. Smith, and H. Boral, "A Single-User Performance Evaluation of the Teradata Database Machine," *MCC Technical Report*, (DB-081-87)(March 1987).

6. G. Graefe, "Encapsulation of Parallelism in the Volcano Query Processing System," *Oregon Graduate Center, Computer Science Technical Report*, (89-007)(June 1989).

7. J. Peterson and A. Silberschatz, *Operating Systems Concepts*, Addison-Wesley, Reading, MA. (1982).

8. G. Graefe, "Volcano: An Extensible and Parallel Dataflow Query Processing System," *Oregon Graduate Center, Computer Science Technical Report*, (89-006)(June 1989).

9. H. Boral, D. DeWitt, D. Friedland, N. Jarrell, and W. Wilkinson, "Implementation of the Database Machine DIRECT," *IEEE*

Transactions on Software Engineering **SE-8**(6) pp. 533-543 (November 1982).

10. M. Stonebraker, "Operating System Support for Database Management," *Communications of the ACM* **24(7)** pp. 412-418 (1981).

11. S. Sechrest, "An Introductory 4.3BSD Interprocess Communication Tutorial," in *Unix Programmer's Manual Supplementary Documents 1*, 4.3 Berkeley Software Distribution (1986).

BIOGRAPHICAL NOTE

The author was born August 30, 1956, in Brewster, Washington. In 1964 he moved from Twisp, Washington to Portland, Oregon. The author graduated from Sunset High School in 1974. He attended Washington State University from 1974-1978, where he graduated cum laude with a bachelor of science degree in architectural studies.

The period 1978-1979 the author worked for a Portland furniture builder prior to opening his own business in August, 1979. The author owned and operated Bortvedt Woodworking (furniture design, manufacture and repair) from 1979- 1988.

In March, 1987, the author began study at Oregon Graduate Center where he completed requirements for a master degree in computer science.

He has been married for seven years to the former Mary Duschik and they have two children, Sarah, age 3, and, Karen, age 2.