APR 16 1982

The person charging this material is responsible for its return to the library from which it was withdrawn on or before the **Latest Date** stamped below.

Theft, mutilation, and underlining of books are reasons for disciplinary action and may result in dismissal from the University.

To renew call Telephone Center, 333-8400

UNIVERSITY OF ILLINOIS LIBRARY AT URBANA-CHAMPAIGN

ENGINEERING

FEB 21 1982

FEB 22 RECD

INTERLIBRARY LOAN

APR 17 RECD

TO REPRODUCTION

OCT 16 RECD

L161—O-1096

Digitized by the Internet Archive
in 2012 with funding from
University of Illinois Urbana-Champaign

http://archive.org/details/researchinnetwor00heal

4
13c
59

Engin.

CONFERENCE ROOM

ENGINEERING LIBRARY
UNIVERSITY OF ILLINOIS
URBANA, ILLINOIS

Center for Advanced Computation

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
URBANA, ILLINOIS 61801

CAC Document Number 209
CCTC-WAD Document Number 6507

Research in Network Data Management and Resource Sharing

Experimental System Progress Report

September 30, 1976

The Library of the
SEP 11 1978
University of Illinois

CAC Document Number 209
CCTC-WAD Document Number 6507

Research in Network Data Management and Resource Sharing

Experimental System Progress Report

by

David C. Healy
Edwin J. McCauley
David A. Willcox

Prepared for the
Command and Control Technical Center
WWMCCS ADP Directorate
of the
Defense Communications Agency
Washington, D.C.

under contract
DCA100-75-C-0021

Center for Advanced Computation
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

September 30, 1976

Approved for release: _____
Peter A. Alsberg, Principal Investigator

Table of Contents

	Page
Executive Summary	1
Nexus	1
Network Virtual File System	2
Query Strategies	3
The Design and Implementation of an Experiment in Distributed Data Mangement	5
Conceptual Foundations	5
Query Execution	7
Component Environment	8
Implementation Considerations	10
Component Architecture	12
Implementation of Nexus	15
References	18
On the Conceptual Design of a Network Virtual File System	19
NVFS Logical Design Overview	20
NVFS Primitives	31
Multi-copy Management	46
References	56
Query Strategy Research	57
Introduction	58
Restrict Prediction	60
Join Prediction	64
Experimental Confirmation	71
An Integer Linear Program Model for Optimization	73
Further Research	75
References	76
Appendix A - Relational Model Terminology	77

Executive Summary

This document consists of progress reports on three activities collectively referred to as the "experimental system effort." They are:

1. a system, "nexus", implemented to explore the feasibility of one class of distributed data management system architectures,
2. the first level design of a network virtual file system (nvfs), and
3. a report on progress in the area of query strategies.

Each of the three sections is relatively independent. For the reader who would like a brief review of the relational model terminology used in the first and third parts, we have included an appendix.

Nexus

Nexus was built to experiment with distributed data management systems built within existing, conventional operating systems. Nexus creates an environment where independent activities are multiplexed onto a single real process. Systems like nexus are appropriate in cases where:

1. real processes are cumbersome and/or expensive to create,
2. multiple independent user requests must be serviced, and
3. the operating system can provide non-blocking I/O, so the I/O wait times of one activity can be used by another activity.

Nexus was successfully implemented at MIT-Multics. Its implementation deliberately avoids any system dependent tricks. Thus, the nexus architecture could be used on other machines. Due to its significant expense, a full distributed data management facility was not created. Facilities were provided to do brute force (i.e., no index use, no optimization) retrievals from moderate size data bases. The communication facilities implemented are closer to a prototype. All communication

uses a simple conceptual mechanism called pipes. Two communicating activities use the same pipe interface whether they are within the same process, in separate processes on one host or are on separate hosts.

The main things learned from the implementation of nexus are

1. the required environment can be created cleanly and efficiently, and
2. transparent, cross-network interprocess communication is feasible.

Network Virtual File System

The network virtual file system is intended to provide transparent access to files located throughout a network. The main features of this initial nvfs design are:

1. a single, easily conceptualized name space supported by network directory services,
2. selective file access,
3. automatic data translation, and
4. support for multiple copies of critical files.

Name space. The nvfs file name space is a tree with no depth limit. The name of an object has two parts, the name of a directory and the name of an entry in that directory. Thus, the full name of an object is a sequence of directory names starting with the root, and ending with an entry name.

Selective access. The nvfs provides an alternative to bulk transfer of files. An nvfs file may be accessed on a record by record basis. Positioning operations allow records to be skipped. Keyed access is also supported. However, indexed sequential access is not supported. The "correct" logical sequence of an indexed sequential file depends upon the collating sequence of the machine. In a heterogeneous network

the collating sequences of some machines can be incompatible with others. Accomodating indexed sequential access is possible. However, the added complexity was sufficient to cause us to omit it from this initial design.

Automatic data translation. By "remembering" the record template(s) of a file, the nvfs can translate the records as they are used. This document suggests only a very modest translation capability, translation between character sets. However, there are no serious barriers to more extensive translation capabilities.

Multiple copies. Ultimately the nvfs should support multiple copies of critical files. Reliability is a "weak link" phenomenon. Data which is critical to file access must also be reliably supported. We have identified some of this data in the nvfs. The most important data are the record cursors which define the record to be affected by the next nvfs primitive. For reliable access, these cursors should be maintained by the user side of the nvfs rather than the server. To minimize the impact of the eventual support of multiple copies, we suggest that the cursors be maintained by the user side even in single copy environments.

The design of the nvfs has served to make some formerly abstract issues (e.g., name space management) far more concrete. It has also focused attention on practical issues like directory services, and failure recovery. The next step is to allow the design to be critically evaluated. Only when the evaluation process is complete is it reasonable to consider prototype implementation.

Query Strategies

In a distributed data management system it is desirable to present the user with a high level interface. That way, the user need

not be concerned with the exact configuration of the system, etc. A given user request can often be satisfied in a number of different ways. The system trys to select the query strategy which would be the optimum under some metric.

Theoretically, this is fairly simple mathematical programming problem. As a practical matter, it is anything but simple. The basic problem stems from the fact that many strategy decisions are based upon how much output a particular part of the query produces. These estimates are unimportant in single-site systems, because they have scant impact on query performance. In a network environment, the size of intermediate results determines, to a large extent, the overall performance. This is because cross-network communication of intermediate results is very slow.

Conventional techniques for estimating intermediate result volumes are inadequate even in simple cases. We have adopted an innovative approach based upon sampling. A small sample of a relation is selected. Queries are run against this sample. The size of the intermediate result from the actual relation can be estimated from the corresponding intermediate result from the sample. In this preliminary report two different sampling strategies are used. Both have strengths and weaknesses. However, the sampling techniques are clearly superior to classical techniques. This is demonstrated in approximately 30 queries run against a 10,000 record file. In each case the sampling techniques accurately predicted the intermediate result volumes.

The Design and Implementation of an Experiment in Distributed Data Management

This section describes the design and implementation of an experimental distributed data management system called "nexus". The goals of the nexus effort were to explore the limits of feasibility of a selected architecture, and to consider some of the problems that would be encountered in designing a more complete system. These goals were met. Due to the expense of creating a complete system, nexus did not progress beyond the feasibility study version. However, it has been successfully used on a continuing basis as part of the query strategies research. In that work, a relation of 10,000 tuples is processed in a brute force way. Response time is acceptable to terminal users.

The following is a "top-down" discussion of the system. To begin, the basic problem of doing relational model queries is presented. Next, we discuss the alternative ways to build a server. One choice is to use a separate real process for each operation currently being performed on that host. We have elected a second choice, that of multiplexing all the operations into a single process. This choice has many impacts on the design which are discussed. Finally, the major features of the actual system are presented.

Conceptual Foundations

The system is based upon the relational algebra.* The user views the data base as a collection of relations which can be operated upon by algebraic operators to produce other relations. By suitably combining these operations, the desired output relation may be obtained.

---

* For a brief review of relational model terminology, please see Appendix A.

To support a more dynamic environment, tuples selected by a query may be modified. This is expressed by adding a modification clause onto the query. Such modification requires synchronization. (This topic is discussed in detail in another report [2].) We have chosen a relational algebra view because the algebraic operators represent actions to be performed. These actions are well defined and easily conceptualized. They lend themselves quite well to direct implementation.

This algebraic view should be contrasted with the user (programmer) interface languages of contemporary, plex-structured data management systems. In these contemporary systems the logical action being performed must be implemented as a sequence of low level actions. It becomes quite difficult to "see the forest". The programmer must be intimately familiar with implementation details such as sort orders and synchronization mechanisms. In the network environment, there will be a large and diverse community of programmers and users. Further, such an environment is likely to be dynamic, as hosts' load and availability change. Incorrect actions caused by an imprecise understanding of a critical detail can have disasterous consequences. Even correct programs can lead to difficult recovery/restart problems due to inconvenient host failures. Many of these problems result from the fact that the "system" cannot "know" what the user intended to do. Thus, to recover, it must back out of all actions performed by the user.

In a relational algebra based system, many of these problems are mollified. The logical action requested by the program is explicit. Its synchronization requirements are defined by this action. Because the entire logical action is specified, it can be run to completion even if the originating host fails. If it is desirable to back out of a partially completed update, this can also be done.

Query Execution

As is discussed in the section on query strategy research, there may be many different possible strategies for performing a given query. Eventually, the operations in the query are assigned to the hosts of the network. If the query is viewed as a tree, the leaves are the actual stored relations. Each of the nodes represents an operation in the relational algebra. These operations must be assigned to hosts, connected logically to the neighboring nodes of the query and initiated. The process of query execution can be viewed as moving tuples up the tree. An operation that has been assigned to a host will be referred to as a component.

The mechanization of the data transfer between components represents a design choice. However, this design choice is fairly easy. It is possible to start all the components at the lowest level of the tree (nearest the leaves), and let them run to completion. Their output (sometimes called hit files) can be transferred to the next level components and so on up the tree. Where two communicating components are on different hosts, the existing file transfer protocol mechanisms can be used in the transfer. When both are on the same host, simpler mechanisms can be used.

A better alternative is to allow tuples to flow, even though the component has not completed. Thus, the output of one component would be immediately available as the input to the next component. Hit files are needed only when all the tuples are needed to perform some operation, such as a sort. (Certain cases of join have similar requirements.)

It is desirable to isolate the communication function. This allows other parts of the system to be implemented using a single

communications interface. A pair of communicating components use the same interfaces whether they are implemented in one process, in two processes on the same host or on separate hosts. The inter-component communication mechanism in nexus is called pipes. (Similar mechanisms have been called "queues" or "mailboxes".)

Component Environment

One of the main functions of the system is to create an environment in which components can be created, destroyed, monitored, and inter-connected. There has been recent interest within the ARPA Network community in the creation of a Network Operating System (NOS) [3]. Such an operating system would provide the environment required. Each of the components would be a task in the NOS. However, an NOS is not an immediate prospect. Nexus was implemented to create and maintain the necessary environment. In operation, each participating host runs nexus as a server. In response to user requests, the various copies of nexus create the appropriate components, inter-connect them, monitor their execution, and destroy them when they are done.

A component has needs similar to that of a process in conventional operating systems. A process is created at the request of a user. The process has its own distinct address space. The process shares resources such as CPU time and data with other processes controlled by the operating system. The operating system monitors the process, keeping track of the resources consumed such as CPU time and real time, etc. The process is deleted either at the request of the user or when it encounters a fatal error (e.g., an address exception).

Operating systems also commonly provide device independent I/O. This relieves the programmer from the burden of device specific I/O by presenting him a uniform interface. In nexus, similar steps were

taken in the area of component to component communications through the use of the pipe mechanisms introduced earlier. All communication between components uses a single interface, whether the two components are within one process, are two processes on the same host, or are on separate hosts. Thus, only the modules which maintain the pipes need be aware of the locations of the communicating components. As evidence of the success of this approach, there were no code changes in the components when cross-network communications were introduced. (Early versions had no network communication facilities.)

There are two possible architectures for implementing the component environment. In the first a process is allocated for each component. The second architecture involves creating a mini operating system which will multiplex many components onto one process.

Single component per process. The server starts a component by directing the host operating system to initiate a real process to execute the component. This architecture has many advantages. The host operating system is responsible for scheduling, address space management, and process creation and deletion. The components are isolated. If one component blocks, the other components can proceed. A component error which causes the process to abort will not harm any other components. However, not all operating systems can support this architecture.

A server must be able to create processes. On many systems (such as Multics and GCOS) process creation is cumbersome. Many operating systems support only a very limited number of process (e.g., 64 in GCOS, about 85 for MIT-Multics) and are often near their limit. If the server ran out of processes throughput would be severely degraded because no new components could be started. Finally, on Multics, process creation is also relatively expensive. This can add substantially to the cost of experimentation.

Multiple components per process. In this architecture, the server acts as a mini operating system, multiplexing the CPU and other resources among the components. Each component has an address space, open files, and pipes managed by the mini operating system. This architecture does not suffer from the limitations of the first architecture. However, the bandwidth of a single process may not be sufficient. A single process can also be stopped by any component.

The operating system must provide non-blocking I/O for this architecture to be viable. If not, any activity which blocks (e.g., a network read) stops all components.

This architecture was chosen for the implementation of nexus at Multics for the following reasons:

1. process are difficult to create and are a scarce resource,
2. non-blocking network I/O can be provided.

Implementation Considerations

The implementation of multiple components per process architectures has requirements similar to those of contemporary operating systems. Let us quickly outline the required capabilities.

Component creation. Nexus starts components at the request of other components. Nexus makes table entries describing the state of the component including next location to be executed, monitoring information, and the address space. A component is deleted by halting it, logging monitoring information, and freeing all resources used by the component.

Block and wakeup. Components, like operating system processes, need a mechanism to manipulate events. A component may need to wait until an event has occurred, or inform another component that an event has occurred.

Scheduling. A component may be in one of three states: blocked (waiting for an event), ready, and running. A running component

can request to block until an event occurs. When the event occurs, the component is made ready. Sometime later a running component will be stopped and put on the ready list so other ready components can proceed.

Stack (address space) management. The state of a blocked (or a ready but not running) component has to be saved until the component is again running. This includes the component's registers and local variables.

Multics provides each process with a stack. Dynamically allocated local variables and registers are stored on the stack. (Static variables are stored in another table.) Each time a procedure is called the registers of the calling procedure are put on the stack followed by the dynamic local variables of the called procedure.

Because there are multiple components being supported by a single process, the use of the stack must be shared. There are two mechanisms to allow components to share the stack. Each component can have its own stack or a component can be forced to remove everything from the stack before it blocks. Giving each component its own stack makes component coding easier because no restrictions are needed. However, this technique would require sophisticated and extremely system dependent mechanisms to manage the multiple stacks. In nexus a component is forced to remove everything from the stack before blocking.

Rescheduling opportunities. As in conventional systems, I/O bound components should be allowed to run when they are ready. Time slicing is the usual approach taken. In a time slicing operating system, once every time interval the running process is halted and another one started. A process using less CPU than the time interval before blocking will proceed near its maximum rate. Nexus uses a

second strategy. It requires components to frequently return to the scheduler.

This second strategy was chosen for two reasons. A Multics process cannot set a timer to interrupt itself while the process is running (or ready). The interrupt will not be seen until the process blocks. Second, if a component can be preempted by another component, data sharing becomes difficult. Critical sections have to be supported.

Static variables. Components share procedures to perform I/O and other functions. In addition, components from several queries may be doing the same thing. Shared procedures are also used here. Some of these procedures have to remember small amounts of information across invocations. For example, a procedure is reading from a file must remember where the file control block is located. These are called static variables.

When the procedure is being used by different components, separate copies of the static variables are required for each component. The chosen solution is to forbid the use of PL/1 static variables. Instead, a work area (state vector) is provided for each component where the equivalent of static variables are stored.

Component Architecture

Each component consists of a number of procedures. The procedures are called by the main part of nexus, do their task, and return. This return provides an opportunity to reschedule another component (i.e., call one of the procedures). It also "empties" the stack, leaving only the scheduler variables on it. Thus, any other component procedure can be called with no problems of stack management.

This architecture for the components leads naturally to their characterization as finite state automata. Each procedure is a state.

Inside the procedure, the next state is determined and passed back to the scheduler. The scheduler transitions a component from one state to the next by calling the appropriate procedure. The scheduler lets a component run for 100 transitions (about 200 milli-seconds) before checking for another ready component. This continues until no ready components are found at which time nexus itself blocks. When an event occurs, nexus readies the component (if any) waiting for that event.

A component can block only at a transition boundary. A component makes a transition by setting variables describing the next state and optional event. If the event is specified, the scheduler blocks the component until the event occurs. Then the component is started at the specified state.

To illustrate this more concretely, we will discuss a component called the relation manager (RM). This component is responsible for retrieving tuples from relations and passing them through the Boolean qualification.

The RM retrieves tuples from a relation and writes the ones satisfying the Boolean criteria to the output pipe. The RM has seven states:

1. init: Parses the request and opens the input file.
2. open_pipe: Opens the output pipe.
3. write_header: Writes a record into the pipe describing the tuples to follow.
4. get_a_tuple: Reads a tuple from the relation.
5. restrict: Tests the tuple to determine if it meets the Boolean criteria.
6. write_tuple: Writes tuples into the output pipe.
7. fatal_error: Gives the user an error message.

The state transitions are specified in a symbolic table. For the relation manager this table is:

RM State Table

Current State	Next State			
	Normal	Block	Error	EOF
init	open_pipe		fatal_error	
open_pipe	write_header	open_pipe	fatal_error	
write_header	get_a_tuple	write_header	fatal_error	
get_a_tuple	restrict	get_a_tuple	fatal_error	terminate
restrict	write_tuple	get_a_tuple	fatal_error	
write_tuple	get_a_tuple		fatal_error	
fatal_error	terminate			

The entries in the next state columns are the return codes describing the next state. For example, if the component has just entered the open_pipe state there are three possible next states. If all goes well and it doesn't have to wait for anything the next state is write_header. If network I/O is required to open the pipe it will probably go to a blocked state waiting for I/O completion. When the I/O completion event occurs the component will again attempt to open the pipe, thus the entry in the block column. Finally, something unexpected may happen, resulting in a transition to fatal_error.

To facilitate easy modification, the table is compiled at each nexus invocation. The ability to modify the transition table has proved very useful particularly during development. If a state is suspected of having bugs, a monitor state can be inserted to check its actions. States can be added to count the number of times a particular transition is made. Dummy states were sometimes substituted for states not completed. The dummy state would simulate the real state. This allowed the development of components needing that state to proceed.

Implementation of Nexus

The global status of nexus and its components is contained in three tables. The first is the component description table containing data on each component. This includes the next state the component will execute, the address of the component's state vector (static variables), and whether or not the component is blocked. Accounting information on each component is also kept in this table. For each component the following are monitored:

1. cpu time used,
2. page faults taken,
3. real time spent running,
4. number of state transitions, and
5. number of times blocked.

The second table, the channel table, contains information on each event channel in use. Associated with each channel is an event counter initially set to zero. When an event is received the counter is incremented. When a component asks to block on the channel, the counter is decremented. If the counter is non-negative the event has been received and the component can continue. Otherwise it must block.

Other information is kept for each channel including:

1. real time blocked on the channel,
2. number of times blocked, and
3. the ID of the component that declared the channel.

The third table is the state monitoring table. For each state the following information can be monitored:

1. CPU time used,
2. real time used,

3. page faults used, and

4. number of times this state was executed.

Six procedures and the scheduler manipulate the component description table. They are:

Initialize. The table is initialized to an empty system.

Create. Create is called with the first state of the component and an optional parameter. An empty slot is found in the component state table and next state is set to the component's first state. A state vector is allocated to the new component. The input parameter is copied into the state vector. The new component is now ready to run.

Remove. The specified component is removed from the system. Any system resources used by the component are freed. (In particular, dynamic memory is freed.) The slot in the component description table allocated to the component is then reinitialized.

Display. The state and monitoring information for the given component are displayed on the user's terminal.

Channel_wait. Channel_wait and channel_post manipulate the component description and channel description tables. These two routines are only used by the scheduler. Channel_wait blocks a component on a channel. The component description table is modified to show that the component is blocked. The channel description table is modified to show the channel on which the component is waiting.

Channel_post. Essentially channel_post does the opposite of channel_wait. Channel_post is used when an event arrives on a channel. The post_count associated with the channel is incremented. If a component was blocked on the channel, the component is marked ready.

Four other procedures exist to manipulate the channel table. Since they are analogous to the first four routines which manipulate the component description table they will only be briefly mentioned.

Initialize. The channel table is set to empty.

Add. A channel is added to the channel table.

Remove. A channel is removed from the channel table.

Display. Monitoring information on all the channels owned by the component is displayed.

Scheduler. The scheduler is responsible for running components. It searches the component description table for a ready component. If one is found the scheduler steps it through 100 states or until it requests to be blocked on a channel.

If the latter case happens, channel_wait is called. Channel_wait marks the component as blocked. In either case the scheduler checks the component description table for any ready components. If none are found, the scheduler does a Multics block on all channels that the various components are blocked on, blocking nexus itself. When the Multics block routine returns, at least one channel has received an event. Channel_post is called to ready some component. Next the scheduler again checks for ready components.

The scheduler does not itself start components. Instead special components are started with the system which handle component creation and destruction. One of these special components "listens" to the terminal controlling nexus. The user can direct nexus to start, display, or terminate components.

References

1. Alsberg, P.A. et al. "Synchronization and Deadlock," CAC Document No. 185 (CCTC-WAD Document No. 6503), Center for Advanced Computation, University of Illinois at Urbana-Champaign, March 1976.

2. Bunch, S.R., Healy, D.C., and McCauley, E.J. "Preliminary Experimental System Design Report," CAC Document No. 170 (CCTC-WAD Document No. 5512), Center for Advanced Computation, University of Illinois at Urbana-Champaign, August 1975.

3. Kimbleton, S.R. "Distributed Computation Study," USC/ISI, March 1976 (available from NTIS as AD-A024670).

On the Conceptual Design of a Network Virtual File Sytem

This section is a first level design of a network virtual file system (nvfs). There are three parts to the presentation:

1. an overview of the nvfs logical design,
2. a presentation of the nvfs primitives, and
3. a discussion of the problems of multiple copy management in an nvfs.

Let us make clear at the outset that there are a great many issues in the design of an nvfs that defy rational evaluation on technical grounds. It has been necessary, even in this preliminary design, to make decisions on many of these issues. In the interests of stimulating discussion we have often tried to include both sides of the issue.

Ultimately, the nvfs should be integrated into the operating and file management systems of the participating hosts. In this case, the user could access network-wide data holdings with no program changes. However, such a task would be extremely expensive. Furthermore, until nvfs experiments are performed, it is too early to freeze a design sufficiently to allow its widespread implementation.

The system proposed in this document is much more modest. We suggest that a limited prototype be built. Such work will allow experiments and analysis which will lead to the design of an operational nvfs. The system discussed here is an attempt to suggest the minimum level of meaningful nvfs support. There are numerous potential extensions or enhancements, only a few of which are explicitly identified here. This is not to suggest that the system is without substantial new capabilities (i.e., capabilities not currently available from services like file transfer protocol). Briefly, three major contributions are made:

1. a generalized network-wide directory service,
2. selective remote file access, and
3. automatic data translation.

NVFS Logical Design Overview

The basic unit of the nvfs is a named file. The user of an nvfs file need not know its physical location. A uniform, easily conceptualized nvfs name space replaces the wide variation of file naming conventions found on the various network hosts. These nvfs names must be interpreted to determine which host(s) have the corresponding real file(s). Then, communication must be established to those host(s). Once established, the communication paths allow nvfs commands to be sent and executed.

NVFS structure. The nvfs is composed of four parts:

1. the user_nvfs,
2. the server_nvfs
3. the logger, and
4. the central directory service.

Every host need not have all four parts.

User_nvfs. The user_nvfs modules are located on the same host as the real user (person) who desires access to nvfs files. They translate his requests into nvfs commands, issue those commands to server_nvfs modules for execution, and interpret the results.

Server_nvfs. The server_nvfs modules actually do the requested file accessing. They must be located on the host where the file is. Logically, a server is dedicated to each open file. Thus, if a person has several files on the same host open, there will be several servers operating independently there. Whether or not these servers are in separate processes is left up to the implementer.

Logger. To begin operations at a host, the nvfs must provide a third component, a logger. The logger listens on a well-known socket as part of the initial connection protocol for the nvfs. When a user_nvfs requires service, it engages in an initial connection protocol with the logger at the desired host. The logger starts up a server and performs the required socket allocation. From then on, the user_ and server_nvfs modules engage in their own protocol. There is no further intervention from the logger. The logger returns to listening on the well-known socket for other service requests.

This formulation of the nvfs is different from the logical organization of the FTP service [6]. In FTP two sets of connections are established. A duplex TELNET connection is used by the user to issue commands and receive responses. The data itself is transferred over another connection. We have adopted the simpler model for two reasons. First, there seems to be no need for the nvfs to provide data access between two remote hosts. (That is, a user on host A initiates nvfs access actions from host B to host C.) Second, the programs using the nvfs will interpret reply codes etc., reducing the need for a TELNET-type of control connection. The required control information can be multiplexed onto the data connection.

Central directory service. The final component of the nvfs, the central directory service, is really a specialized server_nvfs. The function of the central directory service is name interpretation. That is, the central directory service takes an nvfs name and maps it into another name. The complete name interpretation function may comprise efforts by the user_nvfs, the central directory service and the server_ nvfs. The eventual output of the name interpretation process is the identification of the server which has custody of the real file corresponding to the nvfs name, and the information that server needs to open

the file. This information will likely vary from host to host, because operating systems differ in what is required to open a file. We have separated central directory services from server_nvfs because its functions are significantly different from the file access function provided by the server_nvfs. In some ways, central directory service is a misnomer. There is no logical necessity that a single host provide all the central directory services. Indeed, this single host would likely be a severe bottleneck on an operational environment. Rather the "central" is meant to be the opposite of "extreme". The central directory services are those not performed by either the server_ or user_nvfs. It may be desirable that no such central services exist. Rather, all the required interpretation is performed by the user_ and server_nvfs. The exact nature of the directory service function is an important research question. We have included central directory services for completeness.

Nvfs names. The name of an nvfs file can be divided into two parts: the name of a directory, and the name of an entry in that directory. Since an entry may itself be a directory, the naming convention can be applied recursively. Thus, the complete name of an nvfs file is a sequence of names starting from the root name. The resulting directory structure is a tree. There is agreement that entry names should be long, but exactly how long is not resolved. A strong case can be made for no length restrictions, because the restrictions serve no logical purpose. Name length restrictions exist due to implementation considerations. It is also unresolved whether the directory hierarchy should have depth limits. The feeling is that depth limits exist to bound the name interpretation process again as an implementation consideration. Thus, they do not represent a meaningful logical construct.

The choice of this multi-level directory structure is motivated by two factors. First, the multi-level structure is more general

than single-level structures. That is, single-level structures can be incorporated as a special case of multi-level structures. However, this generality would be an unwise choice if it did not serve a useful purpose. This brings up the second factor, isolation. Users typically have several different tasks in various states of completion. The existence of a generalized multi-level directory allows the user to establish sub-directories which contain only the objects of interest for a particular task. These sub-directories allow his file space to be partitioned analogously to the partitioning of his current workload. Sub-directories also allow for reference to composite objects, i.e., an entire sub-tree, e.g., "Print all the files in >udd>Pollux>source_files". A third point is that all major existing system file structures can be accomodated into the nvfs structure. Many of these existing systems (in particular, GCOS) are also tree structures of significant depth.

Notice that this directory structure is completely independent of the locations and physical organization of the files. Contrast this with the RSEXEC [5] structure. In RSEXEC file names are of the form "host_name local_file_name". The host structure adds a level "above" the roots of the local file systems. This seriously reduces the flexibility of the RSEXEC structure. First, the user must know exactly where (at which host) each file is stored. Second, it is impossible to have related files stored at several hosts be logically grouped in one directory. Finally, the RSEXEC directory structure does not adaquately support sub-directories.

Any nvfs object may be assigned additional names. Such additional names can be used as shorthand abbreviations for longer, more descriptive names.

The nvfs supports links. A link is a means for allowing an entry in one subtree to be logically included in another subtree without

being stored there. A link is another nvfs name. Links can be thought of as symbolic pointers to other entries in the nvfs directory structure. They are symbolic because they are re-interpreted each time they are used. Thus, the linked-to entry may be deleted and recreated in between two link uses, and both will work correctly. Links are useful when an entry should be logically included in two distinct subtrees. Rather than wastefully duplicating it in both, a link can be created in one subtree to the entry actually stored in the other. Because a link can refer to a directory, it is possible to include an entire subtree in another one. This can be used to make, say, a library appear in each of several different projects' directories. Links may not be used to create recursive names, i.e., a link that points, perhaps indirectly, to itself. Exactly how this is prevented is not clear. The check cannot be done at link creation time, but must be done as part of the link interpretation process. This is because the link might be legal at creation. Subsequent changes to the directory structure might cause the looping to occur when the link is used.

Protection. The protection mechanisms of the nvfs must be viewed in the light of their intended purpose, to prevent browsing. Since security is a weak-link phenomemon and thus may depend on system factors external to the nvfs, expending significant amounts in providing elaborate nvfs protection facilities is "putting a steel door in a paper wall." If the nvfs is significantly hardened, the potential penetrator will simply chose some other path to the data. In the following discussion of the nvfs protection problem several assumptions are made. First, the network is assumed to be secured. Messages cannot be altered in transit or given a false "return address". Second, the nvfs is implemented on a secure system. No other process can alter the nvfs internal

data. Third, all the parts of the nvfs cooperate benevolently. These assumptions mean that part of the nvfs on one host can trust data sent from another part of the nvfs on a different host. Most of these assumptions can be challenged today. However, the on-going research in operating system security suggests that in the future they will be far less questionable.

The nvfs faces a fundamental problem in its protection mechanism, that of reliably identifying a user. In a single-site file system, the identity of the user is available from system tables. These tables are filled in as part of the login procedure. In the nvfs it is likely that a person will invoke nvfs services on another machine. The server_nvfs must determine the person's identity by cooperating with the user_nvfs where the person is. Such a mechanism is employed in RSEXEC, the BB&N distributed system experiment [5], to avoid what they call the "two login" problem. One could clearly accomplish the identification by requiring the person to login to the server_nvfs whenever services were desired. However, this defeats the transparency which is a major goal of the nvfs.

In the nvfs the required cooperation between the user_ and server_nvfs operates as follows. The person requests the user_nvfs to perform some service (e.g., open an nvfs file). The user_nvfs determines the person's identity from local tables. This local identification is translated into a network-wide unique person name. The translation is required so that the nvfs protection mechanisms can work in a uniform person name space. The user_nvfs then engages in an initial connection protocol with the logger on the nvfs contact socket. The logger initiates a server_nvfs. The user_nvfs then sends the person's nvfs identification to the server. The connection is now associated

with a particular user. In the design of the user_nvfs and server_nvfs care must be taken to ensure that the person-connection association is not broken or altered. This mechanism also relies upon the network control programs in both hosts to retain the integrity of the socket space.

Since the connection is now associated with a particular person, the requests issued over that connection can be validated. The two alternative validation points, the user and server ends, correspond to capability lists and access lists repsectively. In a capability list system each user has a list of accessible objects. When an access is attempted, the capability list is consulted, to see whether the attempt should be permitted or denied. This would mean that the user_nvfs does the checking. If a given person can operate in several different environments (e.g., accounts on several hosts), each environment must contain the appropriate capability lists. There is no simple way to determine which hosts "know about" a particular person. Thus, to change his capability list, we may have to poll the hosts. Finally, if I wish to grant you access to one of my private files, I must write information on your capability list. Although this capability list update would be handled by system procedures, there is still vulnerability here.

An access list system has somewhat more attractiveness. Since the access control information and the object are co-located, the synchronization problem is reduced or eliminated. (When one goes to multiple copies of files, the problems may re-appear, however.) If desired, the access list allows a person to access the object from different environments. There is no problem in determining what access control information needs to be updated when a change is to be made. Finally, to grant you access to my file, I change one of my objects not one of yours. This is inherently somewhat safer.

We have already touched upon the issue of network-wide unique person names. However, this may not be sufficiently general. The access control mechanism must allow the possibility of including identification of the person's environment. This could include such facts as:

1. the host the person is logged on to,
2. his terminal identification,
3. his project and/or account, and
4. identification of the procedure requesting the nvfs service.

The identification of the requesting procedure allows nvfs objects to be manipulated only through pre-specified procedures. Multics presently implements this for two classes of objects, directories and mailboxes. There is sufficient future potential here to include procedure identification in the nvfs protection mechanisms. One does face the problems of remote procedure identification and of the procedure name space definition. These issues are quite similar to the remote person identification problem which introduced this section.

The nvfs has three access types for files: change_protection, read and write. These have the usual interpretations. There is some question as to whether write access implies read access. The more general choice appears to be to place no such restriction. However, this means that care must be taken with the default effects of certain primitives. These are discussed in the next section. There are four access types for directories: change_protection, status, modify and append. Status and modify correspond to read and write respectively. However, only directory access and maintenance procedures may access the directory, hence the separate access types. The third access type, append, allows the user to add entries to the directory, but not to alter existing entries.

In Multics, the access control list for an object is logically part of the containing directory of the object. This is not possible in the nvfs for two reasons. First, the directory and the object may not be co-located. Second, there may be multiple copies of a given directory. Both these two facts lead to possibly troublesome synchronization problems. In the nvfs, the access control list for an object is logically part of that object, and is located where it is.

File structure. An nvfs file is made up of logical records. When the file is created, the user must specify whether the file is to use fixed or variable length records. Fixed length records allow faster and more simple record positioning operations because the offset may be calculated from the desired record number. In effect, the fixed length records allow a "virtual index" to be maintained. Variable length records are more general. There has been internal debate as to whether the only type should be variable length. Since accomodating fixed length records is really not that difficult, the nvfs will support both types. In both cases the length of the record must be explicitly specified for writes, and is returned on reads. This frees the user from needing any knowledge of how a particular host implements its files.

There has also been debate on whether to support stream files, i.e., files with no record structure. The decision was made that such files can be supported with the variable length record structure. The reverse is also true, that a stream oriented system can easily support variable length (or fixed length) records. Since many hosts retain a record orientation, it was felt that the nvfs should also.

Associated with each file opened by the user are two cursors which point to the beginning of logical records. All I/O operations

utilize these cursors. For example, a read operation will read the record pointed to by the next_record cursor, and afterwards advance the cursor to beginning of the next logical record. The second cursor points to the last record accessed and is used to rewrite a modified record. The existence of these cursors allows the user to move around in a remote file, transmitting only selected parts of that file.

As an option, an index may be created for an nvfs file. The index allows the movement of the cursor to a record having a particular key. The key of each record is specified when the record is inserted. Normally, the creation of the index is specified as the file itself is created. When a record is accessed via an index, the next_record cursor is left undefined. Thus, the nvfs does not support indexed sequential files. This decision was made because of the dependence of algebraic comparisons on the collating sequences of the machines. In an indexed sequential file there is a sequence of records with one key followed by a sequence of records with an algebraically greater key. Suppose that there are the following two keys "AB123" and "ABC45". On an ASCII machine "ABC45" is greater than "AB123". On an EBCDIC machine, the opposite is true. Without restricting potential key values, an indexed sequential file physically maintained on an EBCDIC machine will appear to be out of sequence when accessed by an ASCII machine. Because equality comparisons on keys pose no severe probelm in random access files, they can be supported.

Data translation. The major underlying goal of the nvfs is transparency. Thus, it is desirable to automatically perform data translation if it is required. The general data translation problem is one which has had a great deal of research interest in the recent past. However, most data translation projects have had much broader scope than

what we are proposing for the nvfs. The nvfs itself represents a canonical intermediate between the stored representation of the file and the internal representation needed by a particular program. We are proposing only simple translation of fields within a record from their stored representation to their intended form in the target program. Initially, only one kind of translation will be provided, that of moving between character sets. The file will be assumed to be in the native character set of the host where it is stored. The translation will be to the native character set of the host from which the file is being accessed.

The next step in generality is to allow a richer internal structure for each record. A greater variety of transformations would have to be furnished. A definition of internal structure would allow the nvfs file to be more self-defining. It would add a level of error detection which does not now exist in any file system, in that the nvfs could detect when a user assumed an incorrect record format. Incorrect format detection would occur before the user's program aborts due to data inconsistencies. Since the internal structure definition and translation is a longer range project, only preliminary thinking has gone into it. The major unresolved issues are:

1. allowable data types,
2. undefined mappings (e.g., 2**34 is a legitimate single precision integer on Multics or TENEX, but is not on IBM 370's),
3. allowable mapping complexity (i.e., can fields be omitted, can aggregation operations be performed, etc.), and
4. support of multiple record types in the same file.

Another difficulty with generalized internal record structures is that the translation requires knowledge of the data structures used by a

particular compiler on a particular host. For example, consider a two dimensional array. PL/1 and FORTRAN store this array in different ways. The nvfs becomes substantially more complicated if such differences must be accommodated.

NVFS Primitives

Introduction. In this section we present the basic nvfs primitives. For each, we give the name of the primitive, a list of its arguments, and a discussion of its effects. Included in this discussion is a list of possible return codes. Each primitive returns a code which describes its success or failure. We have grouped the primitives as to their function.

Directory Primitives

create_directory

containing directory name

new directory name

return code

Causes the logical creation of an empty directory subordinate to the containing directory name. The creating user is given change_protection, status, modify and append access to the new directory if the creation is successful.

Possible return codes:

operation successful

improper access on containing directory - user requires either append or modify access

containing directory full - it is not clear if this can ever happen

name already exists

containing directory unknown

delete_directory

containing directory

directory to be deleted

real file delete switch

return code

Causes the logical deletion of a directory and all its subtree. If the real file delete switch is "on", then the nvfs will direct the local file systems to delete the files in the subtree. If the switch is "off", the files will be removed from the nvfs, but will still be there, accessible through the native file system(s). This primitive is extremely powerful, and must be used with great care. Its power is unavoidable, since there is no good mechanism for handling "orphans", files and directories under a deleted directory.

Possible return codes:

operation successful

improper access on containing directory - user requires modify access

containing directory unknown

name unknown

not a directory - entry is either a link or a file

create_file

containing directory

new file name

return code

Causes a new nvfs file to be made in the specified directory. Initially, no real file is associated with the nvfs file. This association is done with another primitive. The creating user has change_protection, read and write access to the new nvfs file.

Possible return codes:

operation successful

containing directory unknown

name already exists

containing directory full

improper access on containing directory

associate_real_file

containing directory

nvfs file

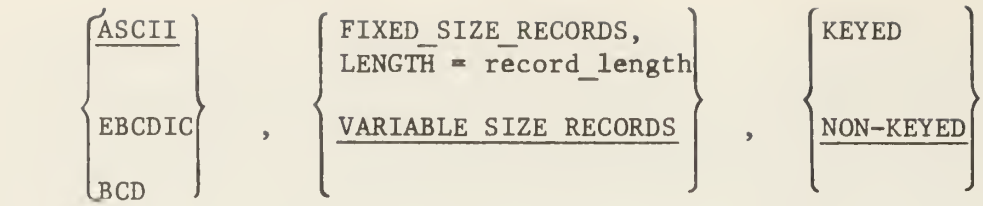
host where real file is located

real file name

real file description

return code

Associates a real file with an nvfs file. The real file may be empty. Eventually, multiple real files could be associated with a single nvfs file by several calls to associate_real_file. This capability would require a great deal of support, and is not planned for the initial nvfs implementation. The user must have write access to the entry. If the nvfs entry did not exist, it is created and a descriptive code is returned. The real file description is intended to serve two purposes. First, it is a means to eventually include the record template information discussed above. Second, the description may include information such as the existence of indices, etc. This second category can be used by servers to determine the most effective way to access a file. In the initial version of the nvfs, the following describes the permitted record descriptions. (Defaults are underlined.)



Possible return codes:

operation successful

containing directory unknown

host unknown - or known not to support server_nvfs

real file name invalid - later feature

real file description invalid

file full - there are already the maximum number of real files associated with the nvfs file

new file created - nvfs file name did not exist, but now does

incorrect access to entry

entry is not a file - one cannot associate real files with nvfs directories or links

delete_file

containing directory

nvfs file name

real file delete switch

return code

Deletes the nvfs file from the nvfs directory. If the real file delete switch is "on", the nvfs causes the host file system(s) to delete the real file(s) corresponding to the entry. If the switch is "off", the entry is still removed from the nvfs, but the real files are not deleted. The user must have write access on the file and modify access to the containing directory.

Possible return codes:

operation successful

containing directory unknown

incorrect access to entry

entry unknown

entry is not a file

disassociate_real_file

containing directory

nvfs file name

host where real file is located

real file name

real file delete switch

return code

Breaks the association of a real file with an nvfs file, and optionally causes the deletion of the real file. The user must have write access to the entry.

Possible return codes:

operation successful

containing directory unknown

entry unknown

incorrect access on entry

real file not associated with entry

entry is not a file

link

containing directory

entry name

linked_to name

return code

Creates a link. The linked_to name is checked for syntax, but not semantics. The user must have modify or append access to the containing directory. The user is given change_protection, read, and write access to the new link.

Possible return codes:

operation successful

containing directory unknown

name already exists

incorrect access on containing directory - user must have modify or append access

bad linked_to name syntax

unlink (delete_link)

containing directory

link name

return code

Logically deletes a link. The user must have write access to the link.

Possible return codes:

operation successful

containing directory unknown

name unknown

not a link

incorrect access on entry

create_real_file

host

real file name

host-specific parameters

host-specific reply information

return code

Asks a host file system to create a real file. Strictly speaking, this is not an nvfs operation. It requires the specific information needed to create a file on a particular host. A parameter is provided for the return of host specific reply information. This primitive is included for completeness.

Possible return codes:

operation successful

host not available

host name invalid

host did not create file

add_name

containing directory

entry name

additional entry name

return code

Adds an additional name to an nvfs object (directory, file, or link). The user must have modify access if the entry is a directory or write access if the entry is a file or link.

Possible return codes:

operation successful

containing directory unknown

entry not found

name already exists - added name duplicates some other name

incorrect access

delete_name

containing directory

name to be deleted

return code

Causes the removal of a name from an object. The last remaining name cannot be removed. The user must have write access if the name is on a file or link. If the name is on a directory, modify access is required.

Possible return codes:

operation successful

containing directory unknown

name not found

last name on object

incorrect access

Protection Primitives

set_access

containing directory

entry name

access mode

nvfs person name

additional constraints

return code

Causes the access control list for the entry to include the specification contained in the primitive. The user must have change_protection access to the entry. The access modes specified must be appropriate for the entry. The additional constraints parameter is presently not used, but is intended to eventually allow a more detailed description of the environment as was discussed above.

Possible return codes:

operation successful

containing directory unknown

entry unknown

incorrect access on entry

invalid access mode

delete_access

containing directory

entry name

nvfs person name

return code

Deletes all access privileges of the person to the entry. The user must have change_protection access to the entry.

Possible return codes:

operation successful

containing directory unknown

entry name unknown

incorrect access on entry

person name not found - this is mostly for information

File access primitives. These primitives are used to access nvfs files. The normal sequence is:

open

{read, write, position, seek}

close

Associated with each open file are two cursors. They point to the last logical record read or written (last_record) and to the record which would be accessed by the next read or write (next_record). Nearly all the primitives both use and alter these cursors. Some operations invalidate one or both cursors. An attempt to use an invalid cursor returns a descriptive code.

open

containing directory

entry name

user logical channel - subsequent operations are in terms of the user logical channel, rather than the longer nvfs file name

open mode - { "read" / "write" / "both" } , { "random" / "sequential" }

sharability - with what classes the file can be shared

"none" - no sharing

"readers" - sharable with other readers (not valid if write or read-write use is specified)

"writers" - this user is willing to share the file with other writers any synchronization is externally provided, a highly dangerous operation. All nvfs guarantees is that a given operation (e.g., read, write) is atomic

time_out - the desired sharing mode must be available within this many seconds or else the open fails

return code

Open initiates the processing needed to access nvfs files. The open process consists of the following steps:

1. name interpretation to determine the appropriate server,
2. connection establishment and identification, and
3. issuance of an open request down the connection.

The server will either accept or reject the open request based on considerations such as security and other users of the file. In any case, a descriptive code is returned. For a successful open, the user_nvfs

sets up a control block and associates it with the user-specified logical channel. All subsequent accesses are made in terms of this logical channel. The logical cursors are set in accordance with the following:

open mode	last_record	next_record
read	null	first record in file
write	null	last record in file + 1
both	null	first record in file

Possible return codes:

operation successful

host not available

invalid mode - either open mode or sharability not valid

logical channel in use

file not closed

desired access mode not supported

containing directory unknown

name not found

incorrect access on entry

not a file - entry was a directory

read

user logical channel

address of (or pointer to) user's buffer

length of the buffer (characters)

number of characters actually read

return code

Reads the logical record pointed to by next_record, translates it as required, and fills the user's buffer with the results. If the file is opened for sequential access, last_record is set to the current value of

next_record, and next_record is advanced to point to the next logical record. If the file is opened for random access, last_record is set to the current value of next_record and next_record is set to null. If the next logical record (i.e., the one to be read) is larger than the user's buffer, no data is transferred, and a descriptive code is returned. There is internal debate over whether the buffer should be filled. This would allow multiple reads using small buffers to access a large record. The problem is that it requires an inter-record cursor. A second problem is that the future extensions in internal record structure (i.e., words) may cause other problems. For example, suppose there were three bytes left in the buffer and the next field is four bytes long. Thus, the decision was made not to support partial record reads (or writes).

Possible return codes:

operation successful

incorrect access on entry - it got changed since the open. (There is debate on whether this is reasonable or not.)

record too long

invalid logical channel - we could use this for signalling any condition which denies access, i.e., server or net failures, access alterations, files not opened, etc.

invalid cursor

end of data - nothing is in the buffer, user has attempted to read beyond the end of the file. Cursors are not changed.

rewrite_record

user logical channel

address of (or pointer to) buffer

length of data in buffer (characters)

return code

Causes writing of the logical record pointed to by last_record (i.e., rewriting it). The length of the data must be equal to the length of last_record. Neither cursor is changed.

Possible return codes:

- operation successful
- invalid logical channel
- record too long
- invalid cursor
- incorrect access on entry

read_length

- user logical channel
- length of next_record (i.e., record pointed to by next_record)
- return code

Returns the length of next_record. This can be used in buffer allocation. If the next_record cursor is invalid, the length is set to -1. (Guaranteed to blow-up most allocators, if the user doesn't check the code.) If next_record is at the end of file, a length of 0 is returned.

Possible return codes:

- operation successful
- invalid logical channel
- invalid cursor
- incorrect access on entry
- end of data

write

- user logical channel
- address of (or pointer to) buffer
- length of data in buffer
- return code

Writes the data in the buffer into the next_record. If next_record is not at the end of the file, the file is effectively truncated, so that the record being written is the last record of the file. If the file has been opened for sequential access, last_record is set after the write to point to the record just written. Next_record is set to point just beyond the end of the file, so that the next write will operate correctly. If the file is open for random access, last_record is set as in sequential access openings. Next_record is set to null. If the file has been opened for fixed length records records of the wrong length are not written and a descriptive code is returned. The cursors are not changed.

Possible return codes:

operation successful

invalid logical channel

incorrect access on entry

invalid cursor

incorrect record length

position

user logical channel

base - "current", "beginning" or, "end": What displacement is relative to. Current means use last_record.

displacement - signed integer number of records

return code

Adjusts cursors by moving a specified number of records from the specified starting point. Next_record points to the desired record, and last_record is set to null. Any combination of base and displacement which results in an effective position that is either beyond the end of the file or in front of the beginning is illegal. There has been

discussion on whether positioning beyond the end of file should be allowed. If it were, complex structures could be built up. Many hosts allow comparable actions. However, the decision was made that in the initial versions of the nvfs, this type of positioning would not be supported. Such an attempted positioning will return a descriptive code, next_record will be set to null, and last_record will be set to the current value of next_record.

Possible return codes:

operation successful

invalid user logical channel

invalid position

invalid cursor - attempted to use "current" but last_record was not valid

seek

user logical channel

key

return code

Causes next_record to point to the record with the specified key. If the file has been opened for write or read/write, this primitive sets the key for the next write operation. As part of that operation (i.e., not as part of the seek) the index will be updated. Duplicate keys are not allowed. This decision was made because it complicates the updating of the cursors if duplicates are allowed. However, this complication is not so severe as to completely preclude the possibility of duplicate keys being allowed in later applications. We have also not included the notion of additional indices. This issue is not clear enough to allow primitives to be specified. Our preliminary thinking is that additional indices can be specified by adding an argument to the seek primitive.

The problem occurs on record insertion. The user would have to have calls to seek for each index so that all the required keys would be set. There is debate on whether a single primitive is more appropriate for this function.

Possible return codes:

operation successful

invalid logical channel

duplicate key

key not found - normal for insertion

no index - operation is not supported

Multi-copy Management

Introduction. This section discusses some of the features need to support multiple copies of nvfs files. Although some of these features have been alluded to before, this topic is sufficiently important to deserve separate coverage. It begins by presenting a conceptual model of multi-copy management. This model is used to illustrate the basic requirements of the system. Next it discusses some design principles derived from the model. To conclude, the nvfs primitives are examined from a multi-copy environment viewpoint.

A model for a multi-copy nvfs. By having multiple copies of selected nvfs files, significant gains in system reliability and availability can be achieved [3,4]. However, considerable effort must be expended to assure that the copies remain consistent. Basically, each copy must receive all updates, and the sequence of updates must be equivalent for each copy. These requirements must be met at all times, including those times when a host with custody of a copy is down. Since the failed host (obviously) cannot process updates during the outage, they must be saved and processed before that copy can be used.

It is desirable that the nvfs mask the existence of multiple copies of its files. The user should not have to "act differently" when a particular file is supported by multiple copies. It is similarly desirable that the nvfs modules be as unaware as possible that they are supporting a multiple copy environment.

Sequencer. To accomodate these requirements, we suggest an extension to the basic nvfs conceptual architecture. The extension adds a functional component, the sequencer, in between the user_nvfs and the server_nvfs. The sequencer is responsible for merging together the streams of primitives from the user_nvfs modules acting for different people using the same file. The sequencer then forwards the composite stream to each of the servers with custody of a physical copy. This mechanism has been also called a primary copy scheme [1,2,3]. The person using a nvfs file with multiple copies actually "talks" to the sequencer rather than to the server_nvfs directly. However, there is no change in the protocol.

Server. Notice that the role of the server_nfvs must change in the multi-copy situation. Each copy must process the update stream in the same order. Thus, the update stream from all concurrent users must be treated as if it came from a single user. In the single copy case each concurrent user of the file had an independent server operating at his behest. If this were continued in the multi-copy case, the nvfs could not guarantee that all copies applied their updates in the same sequence. The sequence for each copy would be determined by contention among the servers. Thus, a single server should process the composite request stream. Since each copy will receive the same stream, these single servers can guarantee that all copies will remain consistent during normal operations. This altered server role introduces other

changes. These are required to prevent user-user interference. They are discussed below.

Flexible pipes. To discuss failure recovery, we introduce a conceptual device. Let each of the communication paths (i.e. user to sequencer and sequencer to server) be a flexible pipe. A flexible pipe has the following properties:

1. messages are inserted in the end,
2. messages may be removed from the beginning (FIFO read) or the end (LIFO read),
3. once inserted, a message does not disappear until it has been explicitly deleted either by the sender or the receiver, and
4. either end may be detached and re-attached without losing any messages.

It is important to emphasize that flexible pipes are a conceptual tool. Their utility lies in the fact that they are a device to understand the necessary actions for recovery. In any nvfs implementation, these recovery actions would have to be performed.

Design principles of a multi-copy nvfs. The nvfs must guarantee that multiple copies of a file remain consistent. During normal mode operation, this is done by the basic model which assures that each copy receives an identical sequence of updates. However, the guarantee of consistency must extend to operations during and after failures. In the following paragraphs we will discuss a variety of failure modes. We will attempt to suggest the design features that the nvfs must possess to accomodate these failures. In general we will consider failures of only the user, sequencer and server components. The flexible pipe mechanism will be assumed not to fail. This seeming rash assumption is really not that unreasonable based on the recent work on

resilient protocols [3]. By correctly sequencing messages and acknowledgments ("ack's"), extremely high reliability communications can be provided.

User failures. Perhaps the simplest failure is that of a user_nvfs. To recover, we insert a "close" in the primitive stream. Such a mechanism assumes that the individual nvfs primitives are atomic and unrelated. That is, the actions performed do not require the use of two primitives to be complete. This is true for simple file applications such as those supported by the basic nvfs. When more complicated actions are required, some mechanism must be provided to group related primitives and execute them collectively. Such collective execution avoids the potentially ugly task of rollback from a partially completed composite action. When composite actions are supported, the "close" must be inserted just behind the last complete composite action. The flexible pipe mechanism can be used to achive this by not deleting any messages in the user-sequencer pipe until a complete composite action has been received by the sequencer. (For increased reliability, the deletion should not occur until after receipt by the servers.) Exactly how composite action bracketing is accomplished is an open issue. For this reason, no attempt was made to include primitives for it.

Server failures. The failure of a server is also handled naturally in this model. The server is no longer there, so messages in that sequencer-server pipe are not deleted, since their receipt is not acknowledged. When the server comes back up, it re-attaches the sequencer-server pipe and begins to process the messages in it. Because requests for exclusive control of the file are also sent via the pipe, the primitives will be processed in a similar exclusive control "state" for all copies of the file. Eventually the failed server will be caught up and

can begin processing current primitives. This catching up does not require any action by the server or the sequencer. The server simply empties the pipe and begins waiting for more input. In systems which do not employ a flexible pipe model, the transition from catch up to normal processing is a very vulnerable point. If failures occur during this transition, they can be rather complex to sort out correctly.

As the server processes messages it acknowledges them back to the sequencer. When the sequencer receives acknowledgments from all the servers it deletes the message from all the pipes in which it was placed. Further, the input message from the user is also deleted. Such a scheme guarantees that messages are available for recovery purposes from two sources, the user and other servers. Care must be taken in the design of the sequencer acknowledgment processing. If this is not done, there may be critical data known only by the sequencer. It is a broad design principle that no such data may exist. If there is data that is critical to the correct functioning of the multi-copy nvfs, access to that data must also be reliable. Normally this would be done through a redundancy scheme, so that critical data is located in several places. The designer then faces the problem of synchronization of the use of this data.

To complete this discussion of server failures, we must mention an additional capability needed by the server: duplicate message elimination. During normal operation the basic mechanisms preclude the possibility of duplicate messages to the server. However, during many recovery scenarios the possibility of duplicate messages exist. The server must eliminate these. In the elimination process the server cannot depend upon a sequencer assigned sequence number. When the sequencer fails the nvfs may not be able to re-establish the sequence

exactly. It is less complex to resubmit all pending user requests in an arbitrary order defined by the new sequencer. The server compares these messages with those already on hand and eliminates any duplicates. Logically, this comparison can be done on the basis of message content. Practically, the use of user_nvfs assigned sequence numbers seems well advised. Notice that the user_nvfs internal variable "last_seq_number" is not a critical data item. If the user_nvfs fails, the system knows the highest message from him that has been received is known. In the recovery a "close" is inserted in the user to sequencer pipe. There is also now no real need for sequencer assigned sequence numbers. However, they are useful for internal bookkeeping purposes.

Sequencer failures. The failure of the sequencer is the most complex single failure. This is because a new sequencer must be established. In the initiation of the new sequencer, the correct "state" of the system must be re-established. As was mentioned above this requires the foresight to have reliable access to any critical variables needed to establish the correct state. So far the only critical data identified are the tables reflecting the acknowledgment status of user messages. These can be reliably maintained if instead of thinking of a single table maintained by the sequencer, the distributed tables discussed in [3] are used.

In the recovery of the sequencer function the first step is to select the host where the new sequencer will be. This can be done in a variety of ways, the details of which are not important here. Now the new sequencer must re-establish the state of the system. First, the new sequencer reconnects to all the user and server pipes. Next, the server pipes must be "balanced". That is, the sequencer must assure that each has the same number of messages in it. An imbalance can occur

due to messages being lost in the network control program of the failed server's host, or because a given message was not inserted in all of the required pipes. This balancing is accomplished by doing LIFO reads on each server pipe. The imbalance is detected, and the messages in the "long" pipe are sent to the "short" one. The next step is to request each user to retransmit any nondeleted requests. As was mentioned above, the requests are only deleted in the user-sequencer pipe when they have been received by a server. This is in keeping with the n-host resiliency concepts contained in [3]. Briefly, an information destroying failure can only occur if n hosts all fail within a critical time. Because the sequence of messages is established in part by their arrival times, the new sequencer may not choose the same composite sequence as the old, failed sequencer. This could happen because the users all retransmit their nondeleted messages. It is possible that the message had been placed in the composite sequence, but not yet acknowledged back to the user for its deletion. Fortunately, the duplicate elimination feature of the servers solves this problem. Basically, those messages already inserted into the composite sequence and received by a server will be eliminated as duplicates by the servers when they are resubmitted by the new sequencer. As was mentioned above, this duplicate elimination should utilize user_nvfs assigned sequence numbers rather than sequencer assigned numbers. After receiving and resubmitting the messages from the users, the new sequencer begins normal operations. Again there is no explicit transition from recovery to normal operation.

There is a question about what to do when the old sequencer becomes available again. One argument is as follows. The old sequencer was selected for some reason, (e.g., most of the traffic originated on that host). This reason is not changed by the failure and therefore the

old sequencer should go back to its former role immediately. This can be accomplished by a simulated failure of the new sequencer. The rules for selecting his replacement will cause the original sequencer to be reinstated.

The argument against such a policy is equally valid. The sequencer is a temporary role. The new sequencer will continue to perform it until either he fails or there are no more users of the file. Simulating a falure does cause extra messages to be sent and extra work to be done. To avoid this extra work, the system will do nothing until a real failure occurs or there are no more users.

Multiple component failures. The failures of multiple components are, of course, more complex than single component failures. The main problem that arises is that multiple component failures seriously impair the system's ability to maintain the characteristics of a flexible pipe. In particular, unless care is taken in the implementation, message loss is unavoidable. At first glance, multiple component failures might seem statistically remote. However, since several components may be actually located on a single host, multiple component failures are not uncommon. Each individual component must participate in the recovery actions discussed above. In addition the normal operation of the system must include activities to allow the recovery to proceed. At present, the resilient protocol mechanisms appear to give us the required capability. However more research is needed.

Nvfs primitives in a multi-copy environment. Most of the nvfs primitives operate on the internal data structures of the nvfs: directories, access control lists, etc. These primitives are logically unaffected by the presence of multiple copies of files. In fact, for some of the primitives the most reasonable implementation strategy is

to use the nvfs file access primitives to implement things like directories. Thus, we will discuss only the file access primitives.

Open. Because open may be used to gain exclusive control over a file it must operate slightly differently in a multi-copy situation. The exclusive control state of the file must be made a reliable data item. Each server needs to know who is in control anyway, making the exclusive control state inherently reliable. During server recovery activity the exclusive control state will not be current, since it is catching up on old updates. However, the flexible pipe mechanism guarantees that the correct sequence of messages is retained. Thus, even though the exclusive control states of the servers may not be identical at any given instant, they will follow the same sequence of states.

Read. There is a fundamental problem with all the actual file access primitives (read, write, rewrite and position). They are context dependent because they depend upon the values of last_record and next_record. Further, each user has its own context. The situation is analogous to a shared disk drive. The effect of a read depends upon where the heads are. It may be desirable to allow highly reliable reads, i.e., a read which can be automatically switched to a backup copy should the primary fail. Further, it is desirable to do this automatically, because if we did not, user programs would require complex internal error recovery mechanisms. There is temptation to suggest that the cursors be made reliable data, i.e., supported with multiple copies. However, this is unnecessary. All that is required is that the location of the cursor be shifted from the server to the user. If the user fails there is no need to recover, since it cannot (obviously) continue to access the file. To achieve this, each request to the server must

include the value of the cursors to be used. Each reply will contain their updated values. This mode of operation also simplifies the server design somewhat. It is no longer necessary to maintain the cursor values for each user. It is an open design question whether this externalization of the cursors is useful or desirable for the single copy case. The initial feeling is that even though externalized cursors are somewhat more complex, the problems associated with supporting two different server_ and user_nvfs configurations are worse.

References

1. Alsberg, P.A. et al. "Preliminary Research Study Report," CAC Document No. 162 (CCTC-WAD Document No. 5509), Center for Advanced Computation, University of Illinois at Urbana-Champaign, May 1975.

2. Alsberg, P.A. et al. "Synchronization and Deadlock," CAC Document No. 185 (CCTC-WAD Document No. 6503), Center for Advanced Computation University of Illinois at Urbana-Champaign, March 1976.

3. Alsberg, P.A., Belford, G.G., Day, J.D. and Grapa, E. "Multi-Copy Resiliency Techniques," CAC Document No. 202 (CCTC-WAD Document No. 6505), Center for Advanced Computation, University of Illinois at Urbana-Champaign, May 1976.

4. Belford, G.G., Schwartz, P.M. and Sluizer, S. "The Effect of Backup Strategy on Data Base Availability," CAC Document No. 181 (CCTC-WAD Document No. 6501), Center for Advanced Computation University of Illinois at Urbana-Champaign, February 1976.

5. Cossell, B.P. et al. "An Operational System for Computer Resource Sharing," Proc. of the Fifth Symposium on Operating System Principles, Nov. 1975, pp. 75-81 (available from ACM, P.O. Box 12105, Church St. Station, New York, NY 10249).

6. Feinler, E. and Postel, J. "ARPANET Protocol Handbook April 1976," NIC Document No. 7104, Network Information Center, Stanford Research Institute, Menlo Park, CA, April 1976.

Query Strategy Research

This section deals with one part of the problem of selecting a strategy to perform a user's query. The user of a distributed data management system should be given a high level interface. This minimizes the impact of changes in the data base organization or the network. The data management system must transform the high level request into a series of lower level actions to be performed at the various hosts. There are often many of these possible strategies for a given high level request. The system should automatically select the optimum strategy. The problem is a lack of accurate data. If the system received the same query twice, it could in theory do the second one optimally. The necessary data would be available from the execution of the first. So, in selecting a query strategy the system must make estimates of the required data.

The types of data needed for distributed query strategy selection differ from those in single-site environments. Because the network bandwidth is low, the amount of data shipped must be minimized. The data being shipped are intermediate results (sometimes called hit files). Accurate estimation of their volume is of critical importance. Very little work has been done previously because estimates of hit file sizes are not very important for single-site cases. What work has been done, has relied upon very questionable assumptions such as uniform distribution of domain values* and independence of domains.

We have devised a novel technique for estimating hit file volumes. A small sample of the data base is created. By running a

---

* For a brief review of relational model terminology, please refer to Appendix A.

query against the sample, the size of the hit files on the actual relation can be estimated. Furthermore, statistical techniques allow the error to be bounded. The initial results from this work have been encouraging. In the test cases run, the sampling techniques performed quite well.

We include the integer linear programming formulation to find the optimum query strategy based on the sampling. Integer linear programs are often computationally unattractive. We have developed a much simpler formulation which achieves the same result. Our method uses two passes over the query tree, and can be shown to find the optimum strategy.

As should be clear there is much work yet to do in the area of query strategies. We are pursuing several promising avenues, including more sophisticated sampling techniques, and strategy selection algorithms.

Introduction

It is clear that a distributed relational data base system will have to perform some kind of query optimization in order to deliver reasonable service. In such a system, one of the major factors in determining cost and response time will be the volume of data shipped over the communications network. Therefore, the optimizer must be able to estimate the volume of output from each operation in the relational algebra query. The sampling techniques discussed in this section should make it possible to make such estimates.

Once volume estimates for the intermediate results and estimates of the costs of performing individual operations are obtained, it is possible to find an optimal query strategy using an integer linear programming model or a bottom-up, top-down tree-traversal algorithm. These deterministic methods based on imprecise inputs will yield a query strategy which might not be optimal when run against the full data base.

One would like to know the "probability of correct guess," or how likely it is that this is actually the optimal strategy. It is also desirable to know the "expected excess cost," or how much loss is expected as a result of not always picking the optimal strategy.

The vast majority of contemporary researchers who attempt to predict the performance of queries assume independence between the domains (fields) of the data base. (See, for example, Hammer [1] and Vallarino [2]. They assume that the selectivity of a simple expression (e.g., "salary > 15000") in a complex query is the same as when the expression appears alone, and does not depend upon the rest of the query. For instance, if 80% of all employees have salary > 15000, and 5% of all employees are janitors, the independence assumption would imply that 80% x 5% = 4% of all employees are janitors with salaries greater than $15000.

This assumption of independence has some major drawbacks. For one thing, it begs the question of how the selectivities of the simple expressions can be estimated. (This is generally done by keeping histories of past queries, or by a priori knowledge of the data base.) Perhaps more important is the fact that there is no quantitative indication of the accuracy of the estimate.

In this section we will discuss a method of query performance prediction which is based on statistical sampling. From the relations of the full data base we will construct sample relations. These sample relations will be small compared to the full relations, so they can be stored at any host which might be called on to optimize a query. Using the samples, it will be possible to extrapolate the performance of a query on the full data base. There will also be quantitative error tolerances associated with the performance estimates.

Strictly speaking, we should make sure that any updates to the data base are reflected in the stored samples. However, the statistical properties of the data base should not change rapidly over time. The samples can be kept, unupdated, until they become poor estimators of query performance. At that time, they can be recreated from the modified data base.

Restrict Prediction

The simplest discussed query consists of a restrict function for each of the two relations, and a binary operation (join, for instance) to be performed on the results of the restricts. We assume that the relations in question are stored on different hosts, so at least one of the intermediate results must be shipped over the network. We will further assume that it doesn't matter where the final result ends up, so the choice of where to do the binary operation depends only on the sizes of the intermediate results. To give an example, we are assuming that the query looks something like $R[B_R] * S[B_S]$. (We use the symbol * to denote a join.) If relation R is stored on host 1 only and relation S is on host 2 only, then the result of $R[B_R]$ should be shipped to host 2 if and only if the size (defined as the number of bytes in a tuple multiplied by the number of tuples) of $R[B_R]$ is less than the size of $S[B_S]$. We write this as $|R[B_R]| < |S[B_S]|$. From now on, for notational simplicity, we assume that all tuples are the same size. Therefore, the volume is just the number of tuples. A tuple size factor could be easily added to the equations if necessary.

Sampling a relation. What is needed is an estimate of P, the proportion of tuples in a relation which satisfy the restrict function. Our approach is to take a random sample of the relation and find p, the proportion of tuples in the sample which satisfy the restrict function.

It should be obvious that p is an estimate of P, but how good is it? A larger sample will give a more accurate estimate, but a quantitive error estimate would be desirable. Yamane [3] gives a formula which relates p, the sample size n, the population size N (i.e., the size of the unrestricted relation), and the precision d. We will omit the lengthy derivation. The formula is:

$$d = \sqrt{\frac{(N-n)\ z^2\ p(1-p)}{nN}} \qquad (1)$$

The constant z is a reliability factor which comes from the normal distribution, and is dependent on the desired "confidence level". For a confidence of 95%, z is 1.96. A z of 3 will give a confidence of 99.7%. A confidence level of 95% (for instance) means that $|P-p|$ will be less than d 95% of the time.

We should note here that this and later formulas for the precision d are not exact, but rather are unbiased estimates. To get an exact value for d generally requires an exact value for some parameter (p in this case) which could only be obtained by searching the entire data base. In this case, using P in place of p would yield an exact value for d. This is clearly impractical, given that P is the quantity being estimated in the first place. The fact that the formula yields an inexact figure, the precision of which is not computed, is generally not bothersome; it is a second-order effect.

Example: Start with a relation containing 10^5 tuples. After taking a random sample of 500 tuples, it is found that 50 satisfy the restrict function. Therefore, p=0.1. Evaluating Yamane's equation (using z=1.96) gives d=0.026. This means that P will be in the interval (0.074, 0.126) 95% of the time. (This phrasing may be a little

misleading. P is a constant. This is really the confidence that the estimate p is accurate within precision d.) Therefore, it would be reasonable to expect 10,000 $\pm$ 2600 tuples from the restrict.

In a practical case, it will be useful to know how large the sample should be to give a desired precision. The equation can be easily inverted to give

$$n = \frac{Nz^2p(1-p)}{Nd^2+z^2p(1-p)}$$

Since n depends on p, it will be necessary to take a small sample to get a provisional value for p. This p can be used to find n. If this n is larger than the original sample size, then that sample can be enlarged to size n. This yields a new value for p, which might necessitate a still larger n. The process can be repeated until the desired precision is obtained. When constructing a semi-permanent sample, p = 0.5 will a worst-case value for n. It should be noted that the equations given are invalid if n is not small compared to N. It appears that N/2 is about the largest reasonable value for n.

Table 1 was adapted from Yamane's book. It shows the required sample sizes for various population sizes and precisions.

Probability of correct guess. Having gotten p_R, d_R, p_S, and d_S for the two relations in the query, one can decide which intermediate result to ship over the net. It would be useful to know the probability that that choice is, in fact, optimal. If $|R|$ and $|S|$ are the sizes of the unrestricted relations, then the estimates of $|R[B_R]|$ and $|S[B_S]|$ are $p_R|R|$ and $p_S|S|$. Assume that $p_R|R| > p_S|S|$. The best strategy then is to ship the restricted relation S. The true values of $|R[B_R]|$ and $|S[B_S]|$ usually will be different from the predicted values, but as long as $|R[B_R]| \geq |S[B_S]|$, the correct decision still will have been made.

Table 1 - Sample size for specified confidence limit and relative precision when sampling in percent, for p = 0.5[a].

Population size	95% confidence interval: Sample size for precision of ±1%	±2%	±3%	±4%	±5%	±10%	99.7% confidence interval: Sample size for precision of ±1%	±2%	±3%	±4%	±5%
500	b	b	b	b	217	80	b	b	b	b	b
1000	b	b	b	375	277	87	b	b	b	b	473
1500	b	b	623	428	305	90	b	b	b	725	562
2000	b	b	695	461	322	91	b	b	b	825	620
2500	b	1224	747	484	332	92	b	b	1249	900	661
3000	b	1333	787	500	340	93	b	b	1363	957	692
3500	b	1424	817	512	346	93	b	b	1458	1003	715
4000	b	1500	842	521	350	93	b	b	1538	1040	734
4500	b	1565	862	529	353	94	b	b	1607	1071	750
5000	b	1622	879	535	356	94	b	b	1666	1097	762
6000	b	1714	905	545	361	94	b	2903	1764	1139	782
7000	b	1787	925	552	364	94	b	3118	1842	1171	797
8000	b	1846	941	558	366	94	b	3302	1904	1196	808
9000	b	1895	953	562	368	95	b	3461	1956	1216	818
10000	4899	1936	964	566	369	95	b	3600	1999	1232	825
15000	5855	2069	996	577	374	95	b	4090	2142	1285	849
20000	6488	2143	1013	582	376	95	b	4390	2222	1313	861
25000	6938	2190	1023	586	378	95	11842	4591	2272	1331	868
50000	8056	2290	1044	593	381	95	15517	5056	2380	1367	884
100000	8762	2344	1055	596	382	95	18367	5325	2439	1386	891
→ ∞	9604	2401	1067	600	384	96	22500	5625	2499	1406	900

[a]This is the proportion of units in the sample possessing the characteristic being measured; for other values of p, the required sample size will be smaller.

[b]In these cases a sample of 50% of the universe will give more than the required accuracy. The formula used in this calculation does not apply when n is more than 50% of N.

The confidence in the decision is merely the probability that $|R[B_R]| \geq |S[B_S]|$ is true. If it is assumed that the two values are normally distributed around the estimated values, then it can be shown that the "probability of correct guess", if relation S is shipped, is

$$F\left(\frac{(p_R|R|-p_S|S|)z}{\sqrt{(d_R|R|)^2+(d_S|S|)^2}}\right)$$

where F(x) is the cumulative, unit-normal distribution function. Sample values of this, assuming that $d_R|R|=d_S|S|$, are presented in Table 2. For instance, suppose $|R|=|S|=10^4$, $p_R=0.100$, $p_S=0.086$, and $d_R=d_S=0.02$, then $d_R|R|=d_S|S|=200$, and $p_R|R|-p_S|S|=1000-860=140$. From the table, one can deduce that shipping the output from S, one will make the correct decision 83.4% of the time.

Expected excess cost. With the above formulas, one can find the likelihood of making the wrong decision. One would also like to know how much will be lost as a result of not always making the right decision. To answer this requires the evaluation of a double integral with no closed-form solution. We have evaluated this numerically, with the results presented in Table 3. The table entries are the number of extra tuples one should expect to have to ship, for given precisions and differences in expected volumes. In the example for Table 2, where $d_R|R|=d_S|S|=200$ and $p_R|R|-p_S|S|=140$, one we would expect, on the average, to ship 12.72 tuples more than necessary (83.4% of the time no more than optimal will be shipped, and 16.6% of the time an average of 12.72/.166=76.63 extra tuples will be shipped.)

Join Prediction

The above random sampling technique will allow prediction of the volume of output from a restrict on a relation. Unfortunately,

$d_R|R| = d_S|S|$ \ $p_R|R| = p_S|S|$

	0	20	40	60	80	100	120	140	160	180	200
10	0.500	0.997	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
20	0.500	0.917	0.997	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
30	0.500	0.822	0.968	0.997	1.000	1.000	1.000	1.000	1.000	1.000	1.000
40	0.500	0.756	0.917	0.981	0.997	1.000	1.000	1.000	1.000	1.000	1.000
50	0.500	0.710	0.866	0.952	0.987	0.997	1.000	1.000	1.000	1.000	1.000
60	0.500	0.678	0.822	0.917	0.968	0.990	0.997	0.999	1.000	1.000	1.000
70	0.500	0.654	0.786	0.883	0.943	0.976	0.991	0.997	0.999	1.000	1.000
80	0.500	0.636	0.756	0.851	0.917	0.958	0.981	0.992	0.997	0.999	1.000
90	0.500	0.621	0.731	0.822	0.891	0.938	0.968	0.984	0.993	0.997	0.999
100	0.500	0.609	0.710	0.797	0.866	0.917	0.952	0.974	0.987	0.994	0.997
110	0.500	0.599	0.693	0.775	0.843	0.896	0.935	0.961	0.978	0.988	0.994
120	0.500	0.591	0.678	0.756	0.822	0.876	0.917	0.947	0.968	0.981	0.990
130	0.500	0.584	0.665	0.739	0.803	0.857	0.900	0.932	0.956	0.973	0.984
140	0.500	0.578	0.654	0.724	0.786	0.839	0.883	0.917	0.943	0.963	0.976
150	0.500	0.573	0.644	0.710	0.770	0.822	0.866	0.902	0.930	0.952	0.968
160	0.500	0.569	0.636	0.698	0.756	0.807	0.851	0.887	0.917	0.941	0.958
170	0.500	0.565	0.628	0.688	0.743	0.793	0.836	0.873	0.904	0.929	0.949
180	0.500	0.561	0.621	0.678	0.731	0.779	0.822	0.859	0.891	0.917	0.938
190	0.500	0.558	0.615	0.669	0.720	0.767	0.809	0.846	0.878	0.905	0.928
200	0.500	0.555	0.609	0.661	0.710	0.756	0.797	0.834	0.866	0.894	0.917
210	0.500	0.553	0.604	0.654	0.701	0.745	0.786	0.822	0.855	0.883	0.907
220	0.500	0.550	0.599	0.647	0.693	0.736	0.775	0.811	0.843	0.872	0.896
230	0.500	0.548	0.595	0.641	0.685	0.727	0.765	0.801	0.833	0.861	0.886
240	0.500	0.546	0.591	0.636	0.678	0.718	0.756	0.791	0.822	0.851	0.876
250	0.500	0.544	0.588	0.630	0.671	0.710	0.747	0.781	0.812	0.841	0.866
260	0.500	0.542	0.584	0.625	0.665	0.703	0.739	0.772	0.803	0.831	0.857
270	0.500	0.541	0.581	0.621	0.659	0.696	0.731	0.764	0.794	0.822	0.848
280	0.500	0.539	0.578	0.617	0.654	0.690	0.724	0.756	0.786	0.814	0.839
290	0.500	0.538	0.576	0.613	0.649	0.684	0.717	0.748	0.778	0.805	0.830
300	0.500	0.537	0.573	0.609	0.644	0.678	0.710	0.741	0.770	0.797	0.822
310	0.500	0.536	0.571	0.606	0.640	0.673	0.704	0.734	0.763	0.790	0.814
320	0.500	0.535	0.569	0.603	0.636	0.668	0.698	0.728	0.756	0.782	0.807
330	0.500	0.533	0.567	0.599	0.632	0.663	0.693	0.722	0.749	0.775	0.800
340	0.500	0.532	0.565	0.597	0.628	0.658	0.688	0.716	0.743	0.768	0.793
350	0.500	0.532	0.563	0.594	0.624	0.654	0.683	0.710	0.737	0.762	0.786
360	0.500	0.531	0.561	0.591	0.621	0.650	0.678	0.705	0.731	0.756	0.779
370	0.500	0.530	0.560	0.589	0.618	0.646	0.673	0.700	0.726	0.750	0.773
380	0.500	0.529	0.558	0.587	0.615	0.642	0.669	0.695	0.720	0.744	0.767
390	0.500	0.528	0.557	0.584	0.612	0.639	0.665	0.691	0.715	0.739	0.761
400	0.500	0.528	0.555	0.582	0.609	0.636	0.661	0.686	0.710	0.734	0.756

Table 2 - Probability of correct guess
when relation S is shipped

$d_R|R| = d_S|S|$ \ $p_R|R| = p_S|S|$

	0	20	40	60	80	100	120	140	160	180	200
10	2.88	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
20	5.76	0.55	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
30	8.64	2.08	0.27	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.00
40	11.51	4.17	1.09	0.20	0.02	0.00	0.0	0.00	0.00	0.00	0.00
50	14.39	6.55	2.43	0.72	0.17	0.03	0.00	0.00	0.00	0.00	0.00
60	17.27	9.08	4.16	1.64	0.55	0.15	0.04	0.00	0.00	0.00	0.00
70	20.15	11.71	6.16	2.90	1.22	0.45	0.15	0.04	0.01	0.00	0.00
80	23.03	14.40	8.35	4.46	2.18	0.97	0.40	0.15	0.05	0.01	0.00
90	25.91	17.13	10.67	6.24	3.41	1.74	0.82	0.36	0.15	0.05	0.02
100	28.79	19.89	13.10	8.20	4.87	2.73	1.44	0.72	0.33	0.15	0.06
110	31.67	22.67	15.60	10.30	6.51	3.93	2.26	1.24	0.65	0.32	0.15
120	34.54	25.46	18.17	12.52	8.32	5.32	3.27	1.93	1.09	0.59	0.31
130	37.42	28.27	20.77	14.83	10.26	6.88	4.46	2.80	1.69	0.99	0.55
140	40.30	31.09	23.42	17.21	12.32	8.58	5.81	3.82	2.44	1.51	0.90
150	43.18	33.91	26.10	19.65	14.46	10.40	7.30	4.99	3.33	2.16	1.37
160	46.06	36.75	28.80	22.14	16.69	12.33	8.92	6.31	4.37	2.95	1.95
170	48.94	39.59	31.52	24.68	18.99	14.35	10.65	7.75	5.53	3.87	2.65
180	51.82	42.43	34.25	27.25	21.34	16.46	12.48	9.31	6.82	4.91	3.47
190	54.69	45.28	37.01	29.85	23.75	18.63	14.40	10.97	8.22	6.07	4.41
200	57.57	48.13	39.77	32.48	26.20	20.87	16.40	12.72	9.73	7.34	5.46
210	60.45	50.98	42.55	35.13	28.69	23.16	18.47	14.56	11.34	8.71	6.61
220	63.33	53.83	45.33	37.80	31.21	25.50	20.61	16.48	13.03	10.18	7.86
230	66.21	56.69	48.12	40.49	33.75	27.88	22.80	18.46	14.80	11.74	9.21
240	69.09	59.55	50.92	43.19	36.33	30.30	25.04	20.51	16.64	13.38	10.65
250	71.97	62.41	53.73	45.91	38.93	32.75	27.33	22.61	18.55	15.09	12.16
260	74.85	65.27	56.54	48.64	41.55	35.23	29.65	24.77	20.52	16.87	13.76
270	77.72	68.13	59.36	51.38	44.19	37.74	32.02	26.97	22.55	18.72	15.42
280	80.60	71.00	62.18	54.13	46.84	40.28	34.41	29.21	24.63	20.63	17.16
290	83.48	73.86	65.00	56.89	49.51	42.84	36.84	31.50	26.76	22.59	18.95
300	86.36	76.73	67.83	59.66	52.19	45.41	39.30	33.81	28.93	24.60	20.80
310	89.24	79.59	70.66	62.43	54.88	48.01	41.78	36.17	31.14	26.66	22.71
320	92.12	82.46	73.50	65.21	57.59	50.62	44.28	38.55	33.39	28.77	24.66
330	95.00	85.33	76.33	68.00	60.31	53.25	46.81	40.96	35.67	30.91	26.66
340	97.87	88.20	79.17	70.79	63.03	55.89	49.35	43.39	37.98	33.09	28.71
350	100.75	91.07	82.01	73.58	65.77	58.55	51.92	45.85	40.32	35.31	30.79
360	103.63	93.94	84.86	76.38	68.51	61.22	54.50	48.33	42.69	37.56	32.91
370	106.51	96.81	87.70	79.19	71.26	63.89	57.09	50.83	45.08	39.84	35.07
380	109.39	99.68	90.55	82.00	74.01	66.58	59.70	53.34	47.50	42.15	37.26
390	112.27	102.55	93.40	84.81	76.77	69.28	62.32	55.88	49.94	44.48	39.48
400	115.15	105.42	96.25	87.63	79.54	71.99	64.96	58.43	52.40	46.84	41.73

Table 3 - Expected excess cost, in tuples,
when relation S is shipped

optimizing more complex queries involving several levels of joins will require estimating the volume of output from join operations. In the following sections we will discuss the case of two relations joined on a single domain.

There are three major variables which determine which of several strategies to use. One is the number of distinct domain values in the joining domain as compared to the number of tuples that would be required in a sample of one of the relations. The other two are the sizes of the respective relations. The eight possibilities are outlined in Table 4, and are discussed in more detail below.

Small number of distinct domain values. If there are only a relatively small number of distinct domain values in the joining domain (no more than a few hundred), then for each relation it can be assumed that either the relation itself is small, or the "multiplicity" (average number of occurrences of each domain value) of the domain is large. If the relation is small, then sampling will not be worthwhile. To get statistically significant results would require sampling a large part of the relation. If the multiplicity is large (implying that the relation is large), a simple random sample can be expected to produce a representative sample of the relation. All of the joining domain values will appear in the sample in "reasonably" large numbers.

As shown in the first part of Table 4, there are three possible situations when joining on a domain with a small number of distinct values. We will elaborate on them here.

The multiplicity of the joining domain may be large in both relations. While this situation is theoretically possible in the relational model, it is not likely to occur in practice. The volume of output produced by such an operation would be very large. If such a

Small Number of Values

	Relation R Small	Relation R Large
Relation S Small	Don't sample	Join random sample of R to full S
Relation S Large	Join random sample of S to full R	Won't happen under normal circumstances

Large Number of Values

	Relation R Small	Relation R Large
Relation S Small	Don't sample	Sample using values of joining domain as basic sampling unit or Join random sample of R to full S
Relation S Large	Sample using values of joining domain as basic sampling unit or Join random sample of S to full R	Sample using values of joining domain as basic sampling unit

Table 4 - Strategies for join volume prediction.

situation does occur, the techniques to be discussed in the next section could be still used, but the required sample size may be unreasonably large.

The multiplicity of the joining domain may be small in both relations. In this case the relations themselves are both small, and sampling is probably not worthwhile.

The multiplicity of the joining domain may be large in one relation and small in the other. In this case it will be worthwhile to sample the larger relation but not the smaller. The system would take a sample of size n from the N tuples in the larger relation and join this sample to the smaller relation after applying the respective restrict functions. If this sample join yields x tuples, then the full join can be expected to yield Nx/n tuples. The precision of this estimate can be expressed as:

$$d = z\sqrt{\frac{s^2(N-n)}{nN}}$$

where

$$s^2 = \frac{\sum_{j}^{n} (x_j - \frac{x}{n})^2}{(n-1)}$$

where x_j is the number of tuples in the sample join which come from the jth sample tuple from the larger relation. Note that $\sum_{j}^{n} x_j = x$.

Large number of distinct domain values. When confronted with the problem of modeling the join between two large relations, one might be tempted to take a simple random sample of each relation, form the join of the samples, and use this sample join to model the actual join. Unfortunately, because each sample will contain only a small fraction of the possible domain values, the sampled join will not "mesh" the same as the actual join. To illustrate the problem, consider joining samples of

100 tuples each from relations of 10,000 tuples which have multiplicities on the joining domain of 1. Each of the 100 tuples in one sample will have only a 1% chance of matching a tuple in the other sample, so only one tuple can be expected from this sample join. This is true even if there are no restrict functions. The full join, without restricts, should have 10,000 tuples, so in effect this is a sample of size 1 from a population of 10,000. This will give a poor precision, indeed. What is needed is a sampling technique which will estimate the output from each relation and also estimate the action of the join.

Toward this end, let us introduce a variation of the sampling technique. In our previous discussions the basic sampling unit was a single tuple. Instead, take individual values of the joining domain to be the basic sampling unit. Before, the output volume was estimated by estimating the probability that each tuple would be included, and multiplying this probability by the total number of tuples. Instead, estimate the number of tuples which will have each domain value and multiply by the number of distinct domain values.* If m of the M possible domain values are picked as the sample, and n_j is the number of tuples with the jth sampled value, then the total number of tuples can be estimated as

$$\hat{N} = \frac{M}{m} \sum_{j}^{m} n_j$$

---

* In cases where the possible domain values are easily identifiable (e.g., employee number or supplier code) then the "number of possible values" will be easy to find. However, when the domain value is a numeric quantity, the number of possible values may be large, particularly when compared to the number of values actually used. If such a numeric domain is to be used in a join (which will probably be an unusual occurrence) it might be necessary to define equivalence classes to join on, rather than try to use the normal equality join. In this case, it is the equivalence classes which must be sampled, rather than the individual values.

The precision of this is:

$$d = zM\sqrt{\frac{s^2(M-m)}{mM}} \qquad (2)$$

where

$$s^2 = \frac{\sum_{j}^{m}(n_j - \frac{\hat{N}}{M})^2}{(m-1)}$$

The equation for d can be easily inverted to yield the sample size m required to produce a given precision when the sample variance is s^2.

To implement this method for predicting join output, first take a sample of the joining domain values. From each relation select and save all tuples which have one of these values. The query in question is run against these samples. The above formulas can then be used to predict the performance of the query when run against the full relations.

There are, of course, disadvantages to this scheme. If the multiplicity and the variance (s^2 in the above equations) are not small, then the number of tuples required for a given precision will be larger than for simple random sampling. Also, because constructing a sample will require searching the relation instead of just picking random tuples, the cost of constructing the sample will be much higher. (The presence of an appropriate index would reduce this extra cost factor.)

Another problem more inherent with the scheme has to do with the fact that the sample is no longer truly random. In particular, if a restrict function mentions the joining domain explicitly, the prediction could be off. There are two ways of coping with this problem. It could be ignored (with some justification) in the hope that the sample will be large enough to smooth out this effect. Alternatively, the query processor could be made smart enough to separate the restrict into parts

that do not depend on the joining domain. These parts could be run separately against the sample and the results combined using an independence assumption. For instance, suppose the join is on domain "location" and the query is "(location = 'London' & stock > 1000) or (location = 'Paris' & stock > 5)." If the sample shows that the "average" location has five tuples with stock > 1000 and ten with stock > 5, then expect 15 tuples to result from this query. It will sometimes happen that the domain value(s) tested in the query will be in the sample. In this case it is not necessary to look at the full relation at all. If, in the above example, both "London" and "Paris" were in the sample, the query could be processed using only the sample. (In light of this, it is tempting to place the most frequently referenced domain values in the sample. This would have to be done with extreme care, however, to preserve the statistical properties of the sample.)

Experimental Confirmation

In order to gain some emperical evidence for the value of sampling, we have run experiments using a test data base. This data base contains 10,000 records with information on fuel storage at military sites. We used two sampling techniques and "classical" independence assumptions to predict the performances of twenty-nine queries. We also ran the queries against the full data base to allow us to observe the accuracy of the predictions. The two sampling techniques used were:

A simple random sample of 100 records. The predicted volume, in records, for the query run against the full data base is 100 times the volume from the query run against the sample. The precision is obtained from equation (1).

A sample by domain value. 100 of the 1553 values of the field called "location" were selected, and the 663 records which had one

Table 5 - Two different sampling techniques as estimators of query performance. Precesions are at 95% confidence level.

Query	Actual Volume Produced	Estimated From 100 Random Tuples	Estimated From 100 Random Locations	Estimated From Classical Assumptions
state=98 (Means a ship)	1752	1900±765	1755±626	109 E
state=uk	190	400±382	93±176	109 E
state=06	701	700±498	994±731	109 E
fuel=145	1015	600±463	1196±338	400 E
fuel=jp4	1227	1400±677	1398±355	400 E
fuel=jp4 | fuel=145	2242	2000±780	2594±620	800 E
command=sac	464	300±333	186±352	86 E
command=mac | command=sac	607	700±498	419±560	172 E
fuel=jp4 & command=mac	27	0±194*	47±88	18 I
reciepts_dod>100000	9	0±194*	31±59	?
atock<10000	8268	8300±733	7812±1167	?
state=98 & fuel=145	55	100±194	124±123	177 I
state=98 & fuel=jp4	4	0±194*	16±29	215 I
fuel=jp4 & stock<10000	638	900±564	668±261	1014 I
stock≤5000 & stock>0	5130	5400±972	4923±1250	?
stock≤5000 & stock>0 & state=98	55	100±194	62±92	899 I
open_inventory<atock	1953	1800±749	2019±538	?
open_inventory>stock & command=sac	239	100±194	109±205	90 I
atate=98 & stock<10000	1608	1800±749	1460±493	1449 I
state=98 & stock<10000 & fuel=jp4	3	0±194*	16±29	178 I
receipts_dod>receipts_comm	1528	1600±715	1600±452	?
command≠sac & command≠mac & fuel=jp4	1110	1400±677	1305±346	1153 I
location>us	2671	2300±821	2794±890	?
location>us & state=98	1740	2000±780	1739±627	467 I
receipts_dod>10	1554	1600±715	1584±472	?
receipts_comm>10	2505	2500±844	2252±665	?
receipts_comm>10 & receipts_dod>10	224	0±194*	248±203	389 I
command=pac/ships | command=pacflt	663	300±333	373±236	172 E
(command=pac/shipa | command=pacflt) & atate=98	583	300±333	311±225	116 I

---

*Actually, since p=0, this ahould be 0±0. We give a more conservative preciaion baaed upon p=0.01.

E - Thia is based on the aaaumption that each diatinct domain value ia equally probable.

I - Thia ia baaed upon indepandance between domains.

of those values were collected. The predicted volume is 155.3 times the volume from the query run on the sample, with a precision obtained from equation (2).

The results of these tests are presented in Table 5. In most cases, sampling predicted the actual query performance well within the theoretical limits. Additionally, the sampling generally yielded better results than predictions based upon classical assumptions.

An Integer Linear Program Model for Optimization

In this section we will outline an integer linear program model which could be used to find a suboptimal query strategy. The global optimum is not guaranteed because permutations in the order of execution of the individual operations are not considered. (Join, for instance, is often an associative operation.) We assume that the query is expressed in a tree format, where each tree node contains a relational operator, that the volume of traffic along each branch is known, and that the cost of performing each operation at each host (assuming the inputs are locally available) is known. The variables and constants which appear in the integer linear program are as follows:

N is the number of operations in the query. The numbering of the operations is considered immaterial, except that operation N is a dummy, final operation which merely collects the resulting tuples.

M is the number of hosts.

J is the host on which the final operation (operation N) should be performed.

V_i is the cost of shipping the output of operation i over the network. ($V_N = 0$)

s_i is the index of the successor to operation i.

E_{ij} is the expense of performing operation i on host j.

x_{ij} is a binary variable with the value one if operation i is performed on host j, and zero otherwise.

t_i is one if the output of operation i must be transmitted over the net and is zero otherwise.

For a given legal assignment of the x's and t's (a legal assignment corresponds to a query strategy), the cost is

$$C = \sum_{i}^{N} [V_i t_i + \sum_{j}^{M} x_{ij} E_{ij}]$$

The integer linear program then, is to minimize C subject to the constraints.

$$\sum_{j}^{M} x_{ij} = 1 \qquad i=1,\ldots,N$$

This assures that every operation is performed exactly once.

$$x_{ij} + x_{s_i k} - t_i \leq 1 \qquad i=1,\ldots,N-1 \qquad j,k=1,\ldots,M \qquad j \neq k$$

This assures that t_i will be one if operation i and operation s_i are performed on different hosts. (t_i will be zero unless forced to 1 by one of these inequalities.)

$$x_{ij} = 0 \text{ or } 1$$

$$t_i = 0 \text{ or } 1$$

$$x_{NJ} = 1$$

It would probably not be practical to use this formulation in a production system. Solution of a large integer linear program can be very expensive. A more practical solution is a two-pass algorithm we have developed. Briefly, the first, bottom-up pass through the tree computes the optimal cost for producing each intermediate operation at each host. The second, top-down pass selects the hosts on which to do the operations.

Further Research

The techniques discussed above still will not handle the most general queries. It would be desirable to be able to construct and use samples when there are three or more relations to be joined on several domains. When a single relation is being joined to two (or more) other relations, each on a single domain, the goal is to be able to take a single sample from each relation such that the samples, when taken together, will allow prediction of the performance of the entire query.

Also, it is now possible to compute the probability of correct guess and expected excess cost only for simple queries where the various volumes can be assumed to vary independently. In the general case this will not be so. For instance, if the volume of one of the inputs to a join varies by a given relative amount, then the volume of the join output will tend to vary by an equivalent relative amount. More advanced techniques should allow computation of these quantities for more general queries.

References

1. Hammer, H., and Chan, A. "Index Selection in a Self-Adaptive Data Base Management System," Proceedings of ACM-SIGMOD Conf. on Management of Data, May 1976.

2. Vallarino, O. "On the Use of Bit Maps for Multiple Key Retrieval," Proceedings of ACM-SIGPLAN Conf. on Data, Salt Lake City, March 1976.

3. Yamane, T. Elementary Sampling Theory, Prentice-Hall, Englewood Cliffs, NJ, 1967.

Appendix A

Relational Model Terminology

(This appendix is derived from the Preliminary Experimental System Design Report, CAC Document No. 170, CCTC-WAD Document No. 5512. Another similar tutorial which also contains a very comprehensive reference list appeared in the March 1976 issue of the ACM Computing Surveys on pages 43 to 66.)

The relational model was proposed as an attempt to improve the theoretical foundations of data base management and to provide a very simple user structure. In the short time since its initial suggestion, the relational model has generated tremendous interest within the academic and research communities.

A relation, R, may be thought of as a table whose columns are labeled. Since the columns (which are called domains) are always referred to by name, their order is unimportant. The number of domains is called the degree of R. The rows of the table (which are called tuples) have the following properties:

1. each tuple is distinct,
2. the ordering of the tuples is insignificant, and
3. the domain values in the tuple are atomic, e.g. character strings, integers, etc. but not structures or repeating groups.

Why has such a simple concept generated such interest? First, because the model is so simple, even the most computer-naive user can understand it. Second, because the relational model has no user-supplied structure, the system designer and data base administrator are free to provide any structure they wish "underneath" the user interface. The data base

administrator is free to index, sort, or cluster a relation in any way he wishes. Further, such structures may be changed at will, with the only impact being on system response times. The user interface is fundamentally an associative view of the data. The user supplies criteria and the system gives back the tuples meeting those criteria.

Previous work on the relational model has led to the definition of two different sets of operators: the relational calculus and the relational algebra. In the calculus the user gives a first-order predicate calculus description of his output tuples. In the algebra the user describes explicitly the operations necessary to achieve a desired output relation. Since it can be shown that the two languages are comparable in their expressiveness, there is no loss of generality by our choice of the algebra for examples.

Only three operators are used in examples: projection, restriction and join. Each takes one or more relations as operands and produces a relation as output. To aid in the definition, examples using the following relations will be presented.

SUPPLIER:

SPLR.NAME	SPLR.#	STATUS	CITY
Ajax	10	9	Tuscola
Acme	20	5	Arcola
Widget	30	6	Mattoon
Bomad	40	8	Urbana

PARTS:

PART.#	PART.NAME	COLOR	WEIGHT
1234	widget	blue	100
1235	widget	green	100
1236	bolt	blue	1
1237	bolt	green	1
1238	nut	purple	1
1239	nut	blue	1

SP:

SPLR.#	PART.#
10	1234
10	1235
10	1238
20	1239
20	1237
30	1234
30	1235
40	1236
40	1238

Projection takes as input a relation and an ordered domain list and produces a relation containing only the domains specified in the domain list. For example, to produce a list of the suppliers and their locations we write:

SUPPLIER [SPLR.NAME, CITY]

which produces the following relation:

SPLR.NAME	CITY
Ajax	Tuscola
Acme	Arcola
Widget	Mattoon
Bomad	Urbana

The domain list is enclosed in square brackets immediately after the relation. Projection may also be used to simply re-order the domains of a relation by including all of them in the projection domain list.

Restriction takes as inputs a relation and a Boolean expression of the domains of that relation. It produces a relation with the same domains as the original input relation. The tuples of the output relation are exactly the tuples of the input relation which satisfy the input Boolean expression. To find the suppliers whose status was greater than 6 we would write:

(SUPPLIER where STATUS > 6)

which would produce the following relation:

SPLR.NAME	SPLR.#	STATUS	CITY
Ajax	10	9	Tuscola
Bomad	40	8	Urbana

Restriction may be combined with projection, both by projection on the input relation or on the output. For example, to find the supplier names and status of suppliers whose status is greater than 6 we write:

(SUPPLIER where STATUS > 6) [SPLR.NAME, STATUS]

or

(SUPPLIER [SPLR.NAME, STATUS] where STATUS > 6)

Either one produces the following output relation:

SPLR.NAME	STATUS
Ajax	9
Bomad	8

As was shown in this last example, many alternative statements exist to do the same thing. This fact allows a variety of optimization techniques to be applied, as is discussed in the section on query strategy research.

The simplest type of join takes as input two relations R1 and R2 that have a common domain D. It produces a new relation whose tuples are formed by concatenating a tuple from R1 to one from R2 with the same value for D. For notational convenience we will assume that D is the last domain of R1 and the first domain of R2. To find the supplier numbers of suppliers who supply green parts we write:

((PARTS where COLOR = green)[PART.#] * SP[PART.#,SPLR.#])[SPLR.#]

The join is denoted by the *. This example produces this output relation:

SPRL.#
10
20
30

The following example may help to show what join does. We will go through join on a tuple by tuple basis.

SP[SPLR.#,PART.#] * PARTS[PART.#,PART.NAME,COLOR,WEIGHT]

taking the first tuple from SP, <10, 1234>, we find the tuple in PARTS that has the same value for PART.#, <1234, widget, blue, 100>. The concatenated tuple <10, 1234, widget, blue, 100> is now output, it is in the output join relation. This process is repeated for each tuple in SP. Of particular interest is the SP tuple <30, 1234>. Semantically this means that both supplier 10 (Ajax) and supplier 30 (Widget) supply part number 1234, the blue, 100 pound widget. Thus, both the following tuples are in the join.

<10, 1234, widget, blue, 100>

and

<30, 1234, widget, blue, 100>

This example eventually produces the following output relation:

SPLR.#	PART.#	PART.NAME	COLOR	WEIGHT
10	1234	widget	blue	100
10	1235	widget	green	100
10	1238	nut	purple	1
20	1239	nut	blue	1
20	1237	bolt	green	1
30	1234	widget	blue	100
30	1235	widget	green	100
40	1236	bolt	blue	1
40	1238	nut	purple	1

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: CAC Document Number 209
CCTC-WAD Document Number 6507

2. GOVT ACCESSION NO.:

3. RECIPIENT'S CATALOG NUMBER:

4. TITLE (and Subtitle): Research in Network Data Management and Resource Sharing - Experimental System Progress Report

5. TYPE OF REPORT & PERIOD COVERED:

6. PERFORMING ORG. REPORT NUMBER: CAC #209

7. AUTHOR(s): D.C. Healy, E.J. McCauley and D.A. Willcox

8. CONTRACT OR GRANT NUMBER(s): DCA100-75-C-0021

9. PERFORMING ORGANIZATION NAME AND ADDRESS:
Center for Advanced Computation
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:

11. CONTROLLING OFFICE NAME AND ADDRESS:
Command and Control Technical Center
WWMCCS ADP Directorate
11440 Isaac Newton Sq., N., Reston, VA 22090

12. REPORT DATE: September 30, 1976

13. NUMBER OF PAGES: 85

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office):

15. SECURITY CLASS. (of this report): UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report)
Copies may be obtained from
National Technical Information Service
Springfield, Virginia 22151

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)
No restriction on distribution

18. SUPPLEMENTARY NOTES
None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Distributed data bases
Network resource sharing
Network virtual file system
Query strategies
Sampling
Multiple copy management
Name space management

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report covers three topics: the implementation of an experimental distributed data management system, the design of a network virtual file system, and recent research in query strategies for distributed data management systems.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUC-CAC-DN-76-209	2.	3. Recipient's Accession No.

4. Title and Subtitle
Research in Network Data Management and Resource Sharing - Experimental System Progress Report

5. Report Date
September 30, 1976

6.

7. Author(s)
D. C. Healy, E. J. McCauley, and D. A. Willcox

8. Performing Organization Rept. No. CAC #209

9. Performing Organization Name and Address
Center for Advanced Computation
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

10. Project/Task/Work Unit No.

11. Contract/Grant No.
DCA100-75-C-0021

12. Sponsoring Organization Name and Address
Command and Control Technical Center
WWMCCS ADP Directorate
11440 Isaac Newton Square, N.
Reston, Virginia 22090

13. Type of Report & Period Covered
Research

14.

15. Supplementary Notes
None

16. Abstracts
This report covers three topics: the implementation of an experimental distributed data management system, the design of a network virtual file system, and recent research in query strategies for distributed data management systems.

17. Key Words and Document Analysis. 17a. Descriptors
Distributed data bases
Network resource sharing
Network virtual file system
Query strategies
Sampling
Multiple copy management
Name space management

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
No restriction on distribution Available from the National Technical Information Service, Springfield, Virginia 22151	UNCLASSIFIED	85
	20. Security Class (This Page) UNCLASSIFIED	22. Price

BOUND BY
HNM

UNIVERSITY OF ILLINOIS-URBANA
510.84IL63C C001
CAC DOCUMENT$URBANA
200-210 1976
3 0112 007263970