The University of Calgary

Distributed DBMS in a Local Area Network

by

Cedric Ping-keung Wong

A thesis
submitted to the Faculty of Graduate Studies
in partial fulfillment of the requirements for the
degree of Master of Science

Department of Computer Science

Calgary, Alberta
August, 1986

ISBN 0-315-32667-0

The University Of Calgary

Faculty Of Graduate Studies

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled, "Distributed DBMS in a Local Area Network" submitted by Cedric Ping-keung Wong in partial fulfillment of the requirements for the degree of Master of Science.

Supervisor,
Dr. Anton W. Colijn
Department of Computer Science

Baecker
Professor Harry D. Baecker
Department of Computer Science

Professor Jim R. Parker
Department of Computer Science

Ron Murch.
Professor Ron A. Murch
Faculty of Management

August, 1986

Abstract

Distributed data base (DDB) systems are a relatively new idea. Although there has been much discussion about DDBs, many problems regarding DDBs still are not very well understood. In general, a DDB system is a piece (or pieces) of complicated software.

DDB systems have become technologically feasible only since the availability of inter-machine communication technology. Before the release of 4.2 BSD UNIX[†], there was no facility in the UNIX operating system that could provide inter-machine communication. For this reason, it was not possible to construct a DDB system on a network of UNIX systems without substantially modifying the UNIX operating system itself.

In this research, we demonstrate a DDB system by actually building a DDB system in a local area network environment. The DDB system employs only those facilities provided by the 4.2 BSD UNIX. No modification to the UNIX operating system is required. This provides us with an opportunity to illustrate a practical DDB system in a local area network environment using the underlying operating system facilities.

In addition, an important aspect of this research is to investigate a few issues concerning DDBs. In particular, we study the following issues:

[†] UNIX is a trademark of Bell Laboratories.

- data replication,
- site autonomy,
- site recovery,
- inter-site communication,
- query processing,
- concurrency control.

As part of the result of this research, a front end system called FREDI was developed. FREDI can couple with a number of INGRES systems and function as a DDB system. INGRES is a relational data-base system developed from 1973 to 1983 at the University of California at Berkeley. Using INGRES frees us from being concerned with the trivial issues of a data-base system, such as data storage problems or data indexing problems. A front-end approach was used because of its generality and because it would allow us to better concentrate on issues regarding data distribution. FREDI supports data replication but not data partition, and accepts a super set of QUEL as its query language.

Our discussion in this thesis will be limited to DDBs based on the relational model, because of the model's simplicity, popularity, and the fact that INGRES happens to be a relational data-base system.

The development of FREDI demonstrates that it is possible to construct a DDB system using a front-end approach that is easy to use, provides much site autonomy, gives good performance to "single-site operations" and reasonable performance to "multi-site operations", and tolerates single site failures.

Acknowledgements

I would like to thank my supervisor, Dr. Anton W. Colijn, for his support and guidance in the production of this thesis. The numerous discussion sessions we had were extremely helpful in gaining insights in dealing with the various kinds of problems encountered. I also would like to thank Dr. Colijn for his kindness and patience.

The manuscript at various stages was proofread by a number of people. In this regard a debt of gratitude is owed to Shirley Baxter, Sam Chaise, Joanne Giesbrecht, and Cathy Turner.

I would like to thank my friend, Murray Mowchenko, who occasionally acted as an "emotional shock absorber" and provided a shoulder to cry on during the quest of this thesis. In addition, I would like to thank Jim and Ruby Shephard, and the "Caring Group" for their emotional support.

I would like to thank the office staff (Dolores, Beverley, Bonnie, Francie, Lorraine, Marion, and Renee) at the Department of Computer Science for their cooperation. In particular, I would like to thank Lesia Hawrelak for her great kindness and understanding. Also, I would like to thank Clarence Comsky for his unfailing ability to listen.

I would also like to thank my family for their encouragement and support. Although separated by great distances, Dad, Elizabeth, Peter, Larry, Wilbert, all of you are an important part of my life.

I would like to thank the University of Calgary for their generosity in

financial support. Without it, this research would not have been possible.

Thank you Heavenly Father for Your never failing love and Your patience. You somehow provided reassurance during times of desperation.

Table of Contents

Chapter 1.

Chapter 2.

Chapter 3.

Chapter 4.

Chapter 5.

Chapter 6.

Chapter 7.

List of Tables

List of Figures

List of Pseudo-Code Listings

CHAPTER 1

INTRODUCTION

Although the idea of distributed computing has been around for many years, it was not until the mid-1970's that distributed data-base (DDB) systems were economically and technologically feasible. Since then, although there has been much talk about DDBs, only a handful of systems have actually been developed. Currently, many of the problems regarding DDBs still are not very well understood. So far, the majority of the developed DDB systems are still at the experimental stage and none has been released commercially. Two of these systems: SDD-1 and R^* (pronounced "R star"), are described briefly in chapter 2.

Since the mid-1970's, the cost of many mini- and micro-computers has dropped to a level where they are affordable to a wide variety of organizations. Meanwhile, the introduction of better and less expensive local area networks (LANs) and data communication common carriers also encourages the development of distributed systems. In the United States, package switching networks such as Telenet, TYMNET and ARPANET [Mart81; Tech80], and similar developments in Canada such as Datapac (Datapac 1000 for short message transmissions, and Datapac 1500 for general data communications) [Mcgl78] and Dataroute, provide reliable and inexpensive communication media for both high volume and low volume data communication requirements.

1.1. WHAT IS A DISTRIBUTED DATA BASE SYSTEM?

We shall briefly define the meaning of a DDB. To date, there has not been a general consensus of what the definition of a DDB should be. Date attempts to define a DDB system as the following [Date83]:

> A distributed database is a database that is not stored in its entirety at a single physical location, but rather is spread across a network of locations that are geographically dispersed and connected via communication links.

Date however points out that the above definition may not be precise. The term "geographically dispersed" to most people generally means machines that are at least a few miles apart from each other, but in reality in certain extreme cases two machines that are conceptually considered geographically dispersed may physically be in the same room.

Based on the difference in the the access rates between "local" and "remote" data, Date adds the following to supplement the definition [Date83]:

> A database is "distributed" if it can be divided into distinct pieces, such that for a given user access to some of those pieces is very much slower than access to others.

In the rest of our discussion we shall use the terms **site** and **node** interchangeably to refer to a physical machine within a DDB system.

1.2. WHY DISTRIBUTED DATA BASES?

There are five main reasons for DDBs:

(1) Performance and storage capacity.

A centralized system has an upper limit in performance and storage capability. In a distributed system, more machines can be added as the needs require.

(2) Total failure resilience.

When a centralized system fails, the complete system fails. When a machine in a distributed system fails, other machines still continue to function. In cases where data are replicated, the system might still be able to provide full access to all of the data after a machine had failed. Otherwise, the system continues to provide services in a degraded mode.

(3) Sovereignty of data.

Computer end users are becoming more and more geographically dispersed. A DDB system can allow a division or a branch of an organization to maintain their data locally.

(4) Flexibility.

A DDB system allows rapid reconfiguration when the application requirements change. Machines can be added, enhanced, and removed when desired. The risk of obsolescence is lowered.

(5) Sharing of data between machines.

Sharing of data is intrinsic in computer systems. However, data sharing between centralized systems is often very clumsy. In a DDB system, data on all participating machines are organized in an integrated, coordinated fashion.

Badal [Bada79] summarizes the reasons as follows:

> Many large, geographically distributed organizations find a centralized database system non-responsive or too costly, or both. Military and computer control systems require the reliability and the availability that centralized data-base systems cannot provide. Moreover, the centralized database system does not allow system extensibility and modularity anywhere near the degree characteristic of distributed data-base systems.

1.3. FACTORS REGARDING DISTRIBUTION

Although distributing a data-base may provide answers to some of the problems encountered in centralized systems, new factors regarding data distribution are introduced. A good DDB system is usually highly modular and therefore is very flexible and very easy to re-organize. However, a badly designed system can become so complex that it may be impossible to control and maintain, and the performance may even be unacceptable.

The following are the major factors concerning distributing a data-base on a network of machines:

(1) Data communication requirements.

The involvement of data communication is inevitable in any distributed system. There are four main aspects regarding data communication:

- communication cost,
- data transfer rate,
- propagation delay,
- network reliability.

(2) Data consistency and integrity control.

Strict enforcement of consistency and integrity controls is required in systems where replicated data are maintained to ensure that all units of data agree with each other. In multi-user systems, updates to data items have to be tightly controlled to avoid "surprises".

(3) Duplication of effort.

The same effort may have to be made repeatedly in different sites.

(4) Security breaches.

A DDB system is subject to more security exposure than a centralized one. It is also more difficult to enforce security controls in a distributed environment. To prevent unauthorized access of data, careful planning and tight control of data are required. Security measures, such as deployment of data encryption schemes, may be used to increase the degree of security in data transmission.

Exactly how the above factors affect the design and the performance of DDB systems still is not very well understood. To gain a better understanding, it is important that more research be done.

1.4. PURPOSE OF THIS THESIS

DDB systems have become technologically possible only since the availability of data communication technology. In the history of the UNIX[†] operating system, before the release of 4.2 BSD UNIX there was no UNIX facility that could easily provide inter-machine communication. For this reason, unless the operating system was modified substantially, it was not possible to build a DDB system on a network of UNIX systems.

In this research, we have taken advantage of the inter-machine communication facilities provided by the 4.2 BSD UNIX and have built a DDB system on a network of UNIX systems. We have demonstrated the development of a practical DDB system that employs only the facilities provided by the operating system. There has been no modification of the UNIX system. This provides us with an opportunity to study the design and the implementation of a DDB system in a local area network environment.

[†] UNIX is a trademark of Bell Laboratories.

In addition, an important aspect of this research is to investigate some of the issues associated with distributing a data-base system. In particular, the following issues are explored:

- data replication,
- site autonomy,
- site recovery,
- inter-site communication,
- query processing,
- concurrency control.

The above issues are not all independent. To support data replication, for example, also directly involves other issues.

As part of the result of this research, a front end system named FREDI (**FR**ont **E**nd for **D**istributing the **I**NGRES system) was developed. FREDI can couple with a number of INGRES systems and function as a DDB system. INGRES is a relational data-base system. By taking advantage of some of the facilities provided by INGRES, it is possible to concentrate our attention on issues of distributing the data-base system. INGRES was chosen because: 1) it is available on UNIX, 2) it is a well developed data-base system.

The prototype FREDI was developed on a network of four machines at the Department of Computer Science, the University of Calgary. Each of these machines is a VAX[†]-11/780, and is connected together by a ring-type network. Each is running a 4.2 BSD UNIX operating system.

FREDI is a homogeneous DDB system in that it couples only with INGRES.

[†] VAX is a trademark of Digital Equipment Corporation.

Our discussion will be limited to DDBs using the relational model only, because of the model's simplicity, popularity, and the fact that INGRES is a relational data-base.

CHAPTER 2

BACKGROUNDS

This chapter provides necessary background materials. We first discuss the logical architectures of DDB systems, followed by a case study of two research DDB developments: SDD-1 and R^*. Finally, an overview of INGRES is presented.

2.1. LOGICAL ARCHITECTURE OF DDB SYSTEMS

In a conventional data-base system, a user refers to a *logical* record. The data-base management system (DBMS) derives the requested data from the *physical* records. In a DDB system the same idea also applies, but now the data might actually be stored in a remote machine and the system might have to find it.

Regardless of how a DDB is constructed, the following basic questions arise: How are the data organized? Where are the data located? Where and how is the mapping between logical organization and physical organization performed?

2.1.1. Front-End Versus Integrated Systems

According to the architecture of the system, DDB systems can be loosely categorized into: 1) **integrated systems** and 2) **front-end systems** [Yu85]. In a front-end system all actual data updates and retrievals are processed by independent data-base system(s). In an integrated system there is no such clear distinction.

Comparing two hypothetical DDB systems side-by-side, functionally equivalent otherwise, one an integrated system and the other a front-end system, an observer is likely to find that the integrated system can provide faster services, and allows easier global controls of data and possibly communication resources [Haas82].

In contrast, the front-end system is more likely to provide higher site autonomy [Haas82]. Furthermore, the front-end system concept makes it possible to form a DDB system from a number of pre-existing DBMSs. These pre-existing DBMSs can simply "plug-in" to the front end system. This is important to many organizations, particularly to those that cannot afford to

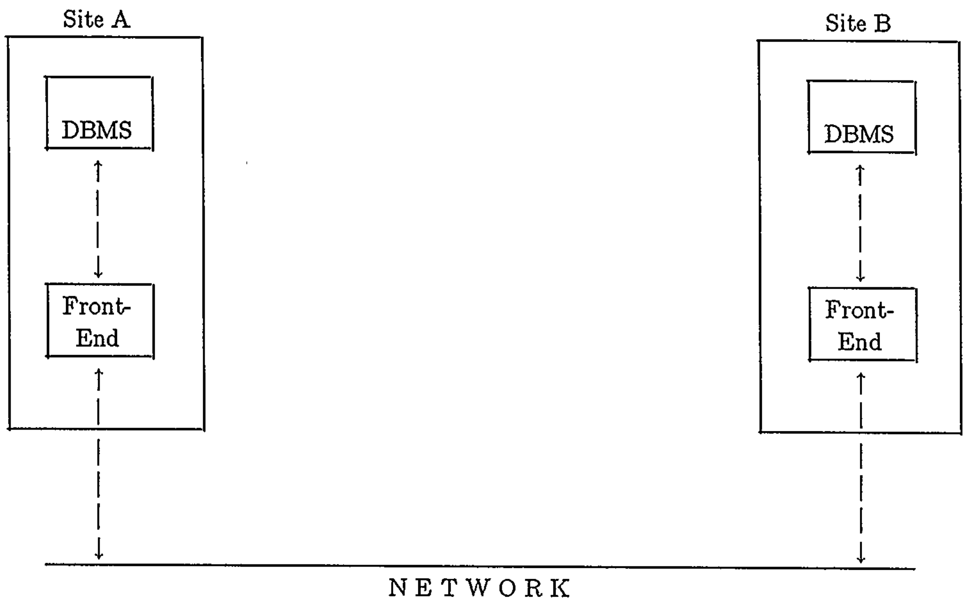


Figure 2.1: The Front-End System Concept.

abandon the data-base systems that are already in use.

From a different perspective, one can regard front-end systems as *loosely coupled* systems, and integrated systems as *tightly coupled* systems [Haas82].

In the rest of our discussion we shall consider front-end systems only, for the following reasons:

(1) To develop an integrated system from scratch would require very serious effort. It would not have been worthwhile to develop an integrated system for the purpose of this research.

(2) To modify an existing data-base system would also involve too much effort, and has potential legal problems.

(3) An important aspect of this research is to investigate issues concerning data distribution (e.g. data replication, site autonomy, etc.), which can be achieved with a front end system.

(4) Front end systems appear to be quite general.

2.1.2. Logical DDB Components

In a typical front-end DDB system, a piece of software (the front end — in this context is known as **network** or **global data manager**) is added to a conventional DBMS (in this context the DBMS is known as **local data manager**) to tackle problems associated with distribution — directory, network interface, conflict avoidance, etc.

The local data manager manages data at its own location and has no awareness of data at other locations or of any issues related to data distribution. The global data manager cannot itself access the data, but relies on the local data manager to read and write the data. It handles all data distribution issues.

The set of data forming the DDB is stored in a number of nodes of which the network is composed. According to their functions, some or all of the following components (each may not exist as a separate process or entity) may be found in a single DDB node [Adib78; Ston76; Spac80]:

- the *local data-base*,
- the *local data-base management system* (the local DBMS),
- the *distributed executive*,
- the *translation function*,
- the *communication adaptation module* (the network handler), and
- the *user process* (or the *user interface*).

The relationships between these components in a hypothetical DDB node is illustrated in Figure 2.2.

When a user or a user program issues a request for data, the input query is first analyzed by the translation function. The distributed executive must then coordinate the processing and response to the user request. Local DBMSs are responsible for retrieving data at local data-bases. Finally, the distributed executive must synthesize all the local retrievals and present to the user or the user process the global response.

The network handler in Figure 2.2 represents a collection of processes and possibly some physical facilities which are necessary to interconnect the nodes. It maintains the knowledge of the physical location of each node, the physical path connections between nodes (at least theoretically), and the protocols to be used in sending messages between nodes.

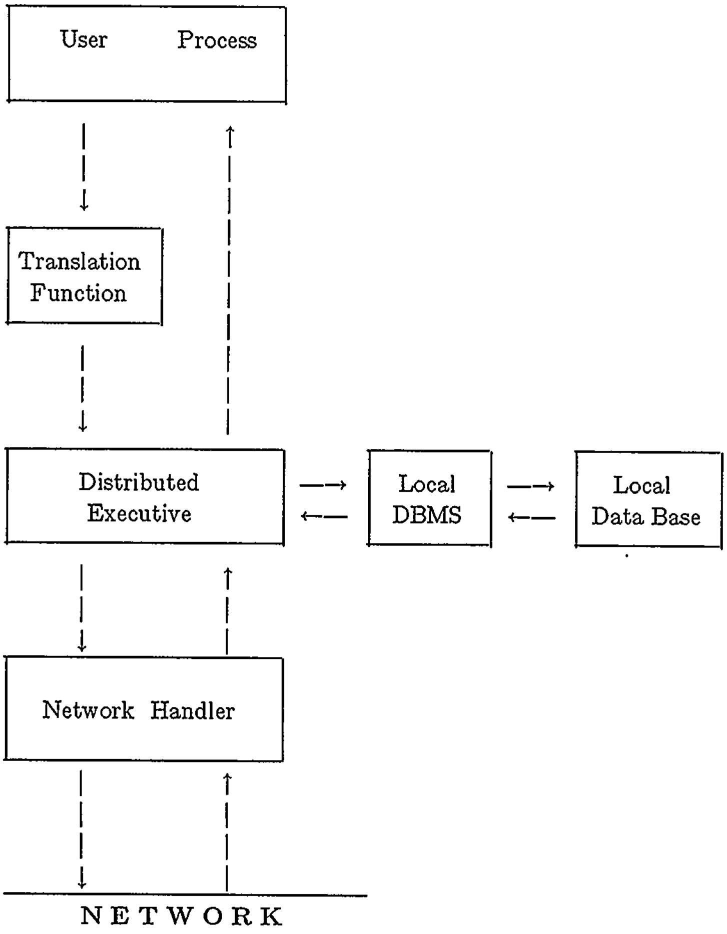


Figure 2.2: Relationships Between DDB Components.

2.1.3. Homogeneous versus Heterogeneous Distribution

If *all* of the local DBMSs in a DDB environment are *identical* or *of the same family,* the distributed system is **homogeneous.** The DDB system is otherwise said to be a **heterogeneous** one [Date83; Draf80]. Booth explains why a heterogeneous DDB is occasionally desirable [Boot79]:

> Probably the most common reason for using unlike computers is that an existing centralized computer system requires expansion. One means of expansion is to acquire a number of minicomputers, place them in point-of-transaction locations, and distribute some of the processing and some of the data-base(s) to those locations.
>
> Even if the minicomputers are acquired from the same vendor who supplied the central processor, they may not be identical to that computer. Often there are good business and/or technical reasons for acquiring the minis from a different vendor, and in that case the likelihood of difference is greater.

A heterogeneous system is likely to be composed of a wide variety of local DBMSs, computers from different manufacturers, and/or computers that use different communication protocols. Consequently, a heterogeneous DDB system is likely to be much more complicated than a homogeneous one.

A very common solution to a heterogeneous DDB is to construct a translator at *each node.* Each of these translators is capable of interpreting back and forth the language (both syntactically and semantically) used by the local DBMS and the language used by the DDB system.

We shall consider only homogeneous DDBs, because:

(1) With a homogeneous system, we can concentrate better on issues regarding data distribution. With respect to the objectives of this research, the serious effort that would be required to construct the translators would not have been worthwhile. However, the construction of such translators that could enable different DBMSs to "talk" to each other could in itself be a very interesting, but different, project.

(2) The 4.2 BSD UNIX provides readily available inter-machine communication facilities.

A number of discussions on heterogeneous DDBs can be found in various writings [Adib78; Boot79; Date83; Draf80; Katz79; Spac80; Taki80; Yu85].

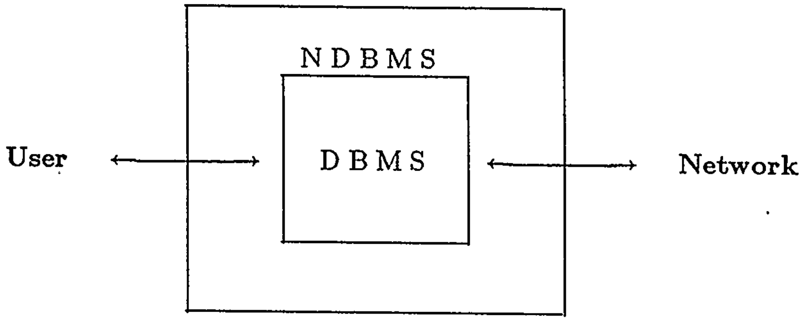


Figure 2.3: CODASYL's NDBMS.

2.1.4. CODASYL's DDB Proposal

Because of its conceptual simplicity and its flexibility to allow participation of pre-existing data-bases, the front-end concept described is preferred by many designers. It is interesting to note that using a similar idea, the CODASYL committee has proposed an extension to their DBMS architecture to enable data distribution of their systems [Mart81]. A new software layer, called NDBMS (Network Data Base Management System), is added to the original CODASYL DBMS (Data Base Management System) structure to perform the network data manager functions. The NDBMS performs the network data management functions. The DBMS manages the local data and has no awareness of any other node.

Quoting from Martin, the NDBMS functions include the following [Mart81]:

- Intercept a user request and determine which nodes to send it to for processing. The majority of user requests should use local data and not require the NDBMS. These may go to the local DBMS directly or be passed to it by the NDBMS.

- Access the network directory (which may possibly be remote) for the above purpose.
- If the target data are on multiple nodes, coordinate the use of these nodes.
- Manage the communication between its node and DBMS's in other nodes.
- If the data-bases are heterogeneous, provide the necessary translation.

There are three possible kinds of nodes in the environment envisioned by the CODASYL committee [Mart81]:

- A user node without a data-base, for example, a minicomputer or intelligent terminal.
- A conventional data-base system without the NDBMS or any cognizance of data distribution.
- A full-function distributed data-base node with the NDBMS.

2.2. OTHER DDB SYSTEMS IN DEVELOPMENT

It is profitable to examine other DDB systems in development. We shall describe two here. They are: SDD-1 from Computer Corporation of America, and R^* (pronounced "R Star") from IBM[†].

2.2.1. SDD-1

The Computer Corporation of America claims to have built the world's first working DDB system — the SDD-1 (System for Distributed Data-bases). SDD-1 runs on a collection of DEC PDP-10s[‡]. It employs ARPANET as its communication network and can also employ X.25 packet-switching networks. It provides full location, fragmentation, and replication transparency. Its query optimizer makes extensive use of the *semijoin* operations [Date83;

[†] IBM is a trademark of International Business Machines.

[‡] DEC and PDP are trademarks of Digital Equipment Corporation.

Date86; Yu85].

SDD-1 assumes CPU cost is negligible; its query optimizer minimizes only the communication cost involved in a query. This is because the data transfer rates of both ARPANET and X.25 are relatively slow and both networks are relatively expensive to use.

SDD-1 was designed for naval command and control applications. It is designed to permit a large amount of replicated data, so as to lessen the amount of data needed to be transmitted across sites during data accesses and to increase the availability and survivability of the information resource, particularly when under military attack. The price for this, obviously, is the high cost of data updates.

SDD-1 updates all copies of the data object immediately; the notion of primary copy does not apply. Concurrency control is based on timestamping, rather than locking, in order to reduce the message overhead. Recovery is based on a *four-phase* commit protocol [Date86], rather than the more commonly known two-phase commit protocol [Date86]. The intent is to make the process more resilient to a failure at the coordinator site.

The treatment of the data catalog in SDD-1 is rather unusual: the catalog is treated just as if it were ordinary user data. The catalog can be arbitrarily fragmented, and the fragments can be arbitrarily replicated and distributed. Since the system has no prior knowledge about the location of any piece of the catalog, a high-level catalog, called the directory locator, is maintained to provide the information. A copy of the directory locator is stored at every site.

SDD-1 handles local issues and distributed issues separately in different modules: The **DM** (Data Module) has the functions of a conventional single site DBMS and has no cognizance of distribution problems. The **TM** (Transaction Module) determines the access strategy for handling distributed data operations efficiently; it does not access the data directly. One TM and up to three DMs can reside at each site. Any node can fail, and the system continues to function. New nodes can be added freely.

No information about the performance of SDD-1 is available.

2.2.2. R^*

The $R^{*\dagger}$ project began in early 1979 at the IBM San Jose Research Laboratory. R^* is an experimental distributed data-base version of the System R relational data-base manager. The design and implementation of R^* were guided by three objectives [Haas82; Lind85; Yost85]:

(1) distribution transparency (i.e. easy to use),
(2) site autonomy,
(3) good (execution) performance.

R^* supports the Structured Query Language (SQL). Applications that use R^* may be written in PL/1, System/370 Assembly Language, or COBOL, by embedding "EXEC SQL" statements which define the data-base requests. A modified version of QMF (Query Management Facility) allows TSO (Time Sharing Option) to make interactive queries. In the current prototype implementation, the granularity of data distribution is a single table (i.e. a relation).

[†] The star in R^* comes from the Kleene Star operator defined by $R^* = ($, R, RR, RRR, RRRR, . . .). It denotes zero or more occurrences of R [Lind80b; Lind80c].

The R^* prototype system is executed within a CICS/VS (Customer Information Control System) system running on the IBM MVS operating system. It consists of multiple cooperating copies of System R. To achieve the distribution transparency objective, R^* provides location transparency; end-users can use SQL just as they do in SQL/DS or DB2 [Yost85]. The current version, however, does not support fragmentation or replication, and therefore no fragmentation or replication transparency either. Consequently, the question of update propagation does not arise.

In order to provide an alternative to true data replication, a new kind of object, called **snapshot**, is invented in R^*. A snapshot is a stored data-base object. It contains a recent copy of some other object(s) which is refreshed periodically by R^*. It is a *read-only* object. The definition of a snapshot is in the form of an arbitrary SQL SELECT query [Date86; Date83; Lind85; Yost85].

To achieve the site autonomy objective, each site in an R^* system is entirely self-sufficient; it does not rely on any form of central service. No site has global knowledge about what operations are taking place at other sites and individual sites will continue to function despite site and/or communication failures [Yost85]. Deadlock detection, recovery, locking, catalog management, and compilation are all performed either *locally* or in a *decentralized* manner. No service is centralized.

One of the tools to achieve good performance in R^* is compilation. In R^*, SQL statements are *compiled* into low level programs called **access modules.** Portions of an access module are stored at each site involved with the execution of a query. During compilation, data-base object names are resolved, access paths are determined, authorization rights are validated, and access plans are distributed to all involved data-base sites. The early binding makes it unnecessary to repeat these time consuming operations each time the application program is actually executed. Since the code was generated at compile time, each site already knows what it should do at execution time. Consequently, the control messages required at execution time are very short: "startup", "stop", "commit", etc. When an access module is created, its dependencies on certain objects are recorded in a system catalog. If one of the depended-upon objects is dropped or its access path is changed, the access module is immediately made invalid and will be automatically recompiled next time it is invoked [Dani82].

Query optimization is performed globally by exhaustive search. The query optimizer minimizes a cost which is a weighted sum of both messages and local processing.

R^* also allows users to physically relocate a table (i.e. a relation). By placing all the needed data in his local system, a user can avoid the overhead of using the communication system when his application is executed. Alternatively, a user can create a snapshot of data.

R^* consists of four primary sub-systems [Dani82; Haas82]:

(1) The **storage sub-system** is concerned with the actual storage and retrieval of data, which are represented as relatively low level objects at a single site.

(2) The **data communication component** provides message passing services.

(3) The **transaction manager** coordinates the implementation of multi-site transactions.

(4) A **data-base language processor** translates programs expressed in SQL to operations provided by the communication and storage system.

Intersite communication between R^* sites is via the CICS Inter System Communication (ISC) facility. CICS-ISC, in turn, makes use of the Virtual Telecommunication Access Method (VTAM) supported by the operating system kernel. VTAM itself is an implementation of the IBM Systems Network Architecture (SNA) communication protocols. Besides providing the facilities to establish and communicate over virtual circuits between processes at different sites, VTAM/SNA also provides support for detecting the loss of communication between sites. The network protocols will notify the end points of the virtual circuit of any process, processor, switching node, or communication line failures. Thanks to this capability, the requesting site is informed when an awaited response will not be forthcoming due to a site failure or a link failure; time out is not needed and is not used in R^*.

In R^*, there exist two kinds of object names: **print-name** and **system-wide name.** Every object has a unique system-wide name, a four-tuple identifier, of the form:

<CREATOR_SITE>@<CREATOR_NAME>.<OBJECT_NAME>@<OBJECT_BIRTH_SITE>

The <CREATOR_SITE> and <OBJECT_BIRTH_SITE> components of the system-wide name are R^* data-base site names whose uniqueness is guaranteed by measures taken outside of R^*. <CREATOR_NAME> is a user name which is unique at a particular site.

A print-name is a simple unqualified name that users normally use to refer to an object. It is either the <OBJECT_NAME> component of the system-wide name, or a synonym for that system-wide name. R^* resolves print-names using synonym tables for each user and name completion defaults.

At each site, a local catalog entry for each object for which it is the *birth site* is maintained. Each site also maintains a local entry for each object for which it is the *current site*. Any inquiry about an object is first directed to its birth site. If the current site of an object is different from its birth site, the entry of the catalog at the object's birth site will point to the object's current site. Any object can be located in at most two remote accesses [Dani82; Date86].

R^*, therefore, does not truly support location transparency *per se*. When a print-name is given, R^* simply performs a table lookup for the corresponding system-wide name. It acts very similarly to an "alias" naming system. At the beginning, the user still has to know at which site the data object was born, the name of the creator, and the site where the creator is located.

It is interesting that at the early stage of the R^* project, both data replication and data partition were under consideration [Bert83; Dani82; Haas82]. However, the prototype R^* supports neither data replication nor data partition [Date86; Lind85; Yost85]. The reasons behind this decision perhaps may best be explained by Lindsay [Lind85]:

> Data partitioning and replication are both quite complex problems with serious implications for query optimization and execution strategies. The distributed query optimization and execution facilities are near the limits of manageable complexity. The implementation of snapshots, a limited form of replication, brought us even closer to the complexity limit barrier. While we feel that partitioned and replicated data are important and useful features of a distributed data-base management system, we realized that a major effort would be required to implement such support.

2.3. AN OVERVIEW OF INGRES

Before the discussion of the development of FREDI (**FR**ont **E**nd for **D**istributing the **I**NGRES system), it is valuable to have a close look at the underlying INGRES system. This section highlights some of the important aspects of INGRES.

INGRES (**IN**teractive **G**raphics and **R**etrieval **S**ystem) [Ston76] is a relational data-base system developed to run within the UNIX operating system environment. A significant portion of the system is written in the programming language C. The parser for input commands is generated with the help of YACC [Ston76]. The primary query language is QUEL (**QUE**ry **L**anguage). Actual retrieval of data is done with the help of the AMI (Access Method Interface) language [Ston76].

2.3.1. The Basic INGRES Process Structure

The INGRES process structure constitutes four processes, each carrying out different tasks. Processes are initiated by means of a sequence of UNIX operations called **fork** and **exec** [Ston76]. The operation fork "splits" the core image of a program into two copies, while exec "overlays" the existing program with a new one [Kern78; Kern84]. Communication between processes is achieved by a message passing mechanism in UNIX called **pipe.** A pipe is a single directional communication. Information written by one process at one end of the pipe can be picked up by a second process at the other end [Kern84; Ston76; UNIX84]. A pictorial representation of the INGRES process structure is shown in Figure 2.4.

In INGRES, each process can communicate only with the *adjacent* processes [Ston76]. The corollary of this is the simple control flow within the INGRES processes. Commands are passed between processes in one direction only, while results and error messages are passed in the opposite direction.

Also, it follows from the use of pipes that all four processes are fully synchronized. This implies that no parallel processing is taking place. At any one time, only one of the four processes can be *in action* — the other three remain idle.

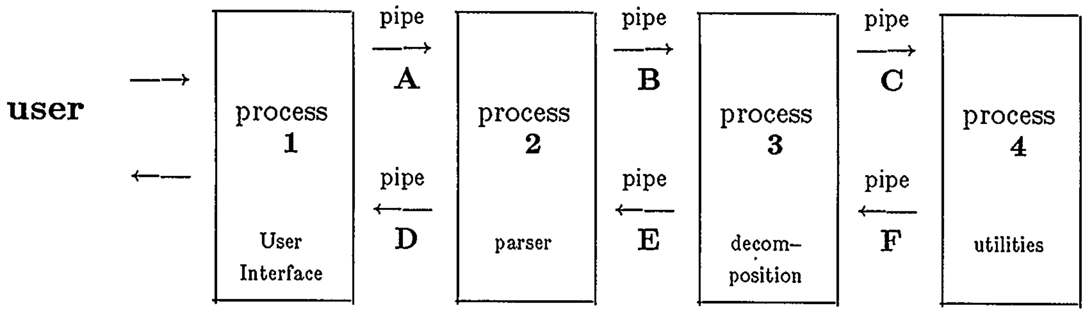
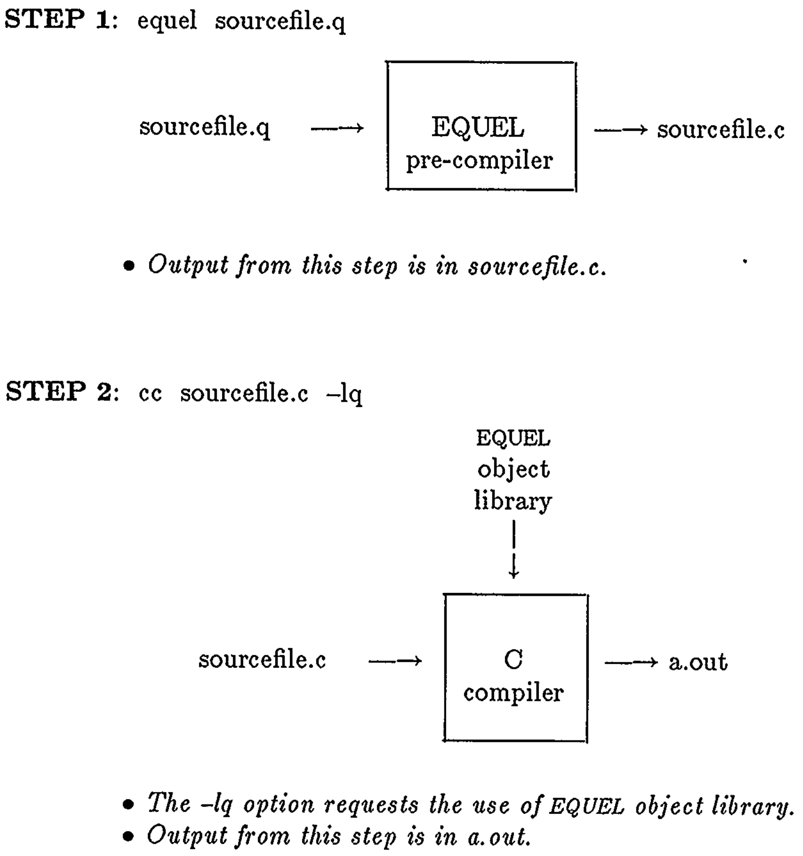


Figure 2.4: INGRES Process Structure (After [Ston76]).

2.3.2. Interface to INGRES

There are two ways a user can communicate with INGRES. He can either: 1) employ the interactive terminal monitor provided by INGRES and issue QUEL statements interactively, or 2) with the aid of the EQUEL (Embedded QUEL interface to C) pre-compiler [Wood79], embed QUEL statements in a C program to provide his own interface. The interactive terminal monitor or



STEP 1: equel sourcefile.q

- *Output from this step is in sourcefile.c.*

STEP 2: cc sourcefile.c –lq

- *The –lq option requests the use of EQUEL object library.*
- *Output from this step is in a.out.*

Figure 2.5: Interfacing INGRES Using a User Supplied C Program.

the user supplied C program is represented as process **1** in Figure 2.4.

The interactive terminal monitor provides the user with a convenient means to communicate with INGRES. It maintains a workspace (a temporary work-file) which contains the would-be QUEL commands entered by the user. The user is free to edit the contents of the workspace. When the user is satisfied with the editing of the workspace, he signals INGRES (by issuing the *go* command) to execute the queries he has entered. When this occurs, the contents of the workspace are passed down to the next process (process **2** — the command analyzer) as a stream of characters through pipe **A**.

Should a user decide to supply his own C program, he first pre-compiles the program with the EQUEL pre-compiler, then compiles the output with the C compiler. An outline of this procedure is found in Figure 2.5.

It should be mentioned that there is no performance gain using a program constructed with the aid of EQUEL versus invoking INGRES interactively. Queries from the program are still interpreted as if they were ad hoc statements. Parsing and finding an execution strategy are done at run time, interaction by interaction [Ston76].

2.3.3. Command Parsing, Integrity and Concurrency Control

Process **2** contains four main components:

(1) a lexical analyzer;
(2) a parser (generated with the help of YACC);
(3) concurrency control routines
(4) query modification routines.

When the would-be QUEL commands arrive at process **2**, they are immediately analyzed by the lexical analyzer and the parser. If everything in the query is legal, a tree structure representation of the query is passed to the next process (process **3**) through pipe **B**. If errors are detected in the input stream, error messages are passed back to process **1** through pipe **D**, and no further action will be taken by process **2**.

Whenever necessary, integrity constraints are imposed on queries at this time. The integrity control routines impose integrity constraints by actually augmenting the QUEL commands. The required integrity control information is stored in the data-base in the VIEW, the INTEGRITY, and the PROTECTION system relations. As an example, suppose that a user, who is allowed to access information about only those employees who are making less than $30,000 a year, has issued the following queries:

```
range of e is employee
retrieve (e.all) where e.name=“Brown”
```

The queries will be modified as this:

```
range of e is employee
retrieve (e.all) where e.name=“Brown”
                 and e.salary<30000
```

Notice that the required qualification(s) is ANDed onto the user’s interaction.

The concurrency control routines are responsible for the control of data consistency in the data-base in a multi-user system. The purpose is to ensure the serializability[†] of transactions. In the current INGRES implementation, a single QUEL command is defined as an **atomic unit** or a **transaction.** This implies that data consistency is guaranteed within a single QUEL statement.

To support a single QUEL statement as an atomic unit is relatively simple., according to Stonebraker [Ston76], one of the INGRES designers. It

---

[†] If a number of transactions that are executed concurrently are serializable, this means the the final result would be the same as if the transactions are executed one after the other, one at a time, in some order [Adib78; Date83; Lann80; Mart81].

can be achieved by physical locks on data items, pages, tuples, domains, relations, etc. Alternately, it can also be achieved by predicate locks. Currently, it is done by physical locks on domains of a relation. Deadlocks are prevented rather than detected: this is done by not allowing an interaction to proceed to process **3** until it can lock all required resources.

The possibility of supporting scopes other than a single QUEL statement as an atomic unit has also been considered. The analysis is as follows: If something smaller than a QUEL statement is chosen as an atomic unit, then the result from the execution of two QUEL statements may not be repeatable. This is obviously undesirable. For something larger than a single QUEL statement, there are three alternatives:

(1) A collection of QUEL statements with no intervening C code.

(2) A collection of QUEL statements with C code but no system calls (such as **fork** and **exec**).

(3) Any arbitrary EQUEL program.

Stonebraker [Ston76] expresses that choice 3) would be impossible to implement. If transaction T2 is **fork**ed from transaction T1, and both T1 and T2 are running concurrently, it is possible that the update in T2 may conflict with the update in T1. Prior to the **fork**, there is no way to tell the conflict between T1 and T2, since the form of update in T2 is not known in advance. Choice 2) is achievable, but it would be very complex to do so. The concurrency control routines would be very difficult to write. Moreover, it would be difficult to enforce in the EQUEL translator unless the translator parsed the entire C language. Choice 1) can be implemented relatively easily, but Stonebraker maintains that due to the lack of requests from users, it has not been implemented.

2.3.4. Query Processing

Query commands in QUEL include four commands: RETRIEVE, REPLACE, DELETE, and APPEND. Process **3** is essential to the processing of these commands.

Any update command (REPLACE, DELETE, or APPEND) is first turned into a RETRIEVE command in order to isolate the tuples to be changed. Copies of modified tuples are stored in a temporary spool, and the actual update is deferred to be processed by the **deferred update processor** in process **4**. Other commands, such as CREATE, DESTROY etc. are classified as *utility commands* and are simply passed along to process **4** without any processing.

The evaluation of a query is achieved by alternating between the following procedures [Ston76; Wong76]:

(1) The *reduction* procedure decomposes a query involving more than one variable into a sequence of sub-queries, each with one or no variable in common. Partial results are accumulated until the entire query is evaluated.

To explain this, the author borrows the example given by Wong [Wong76]. Consider a data-base with three relations:

Supplier (Snum, Sname, City)
Parts (Pnum, Pname, Size)
Supply (Snum, Pnum, Quantity)

and a Query **Q**:

```
range of (S, P, Y) is (Supplier, Parts, Supply)
retrieve (S.Sname) where
                        (S.City = 'New York')
                and     (P.Pname = 'Bolt')
                and     (P.Size = 20)
                and     (Y.Snum = S.Snum)
                and     (Y.Pnum = P.Pnum)
                and     (Y.Quanity > 200)
```

The first detachment might be a restriction on **Parts** in **Q** being replaced by:

Q1:

```
range of P is Parts
retrieve into Parts1 (P.Pnum) where
                        (P.Pname='Bolt')
                and     (P.Size=20)
```

Q':

```
range of (S, P, Y) is (Supplier, Parts1, Supply)
retrieve (S.Sname) where
                        (S.City='New York')
                and     (Y.Snum=S.Snum)
                and     (Y.Pnum=P.Pnum)
                and     (Y.Quantity>200)
```

Note that the variable P.Pnum is common to both Q1 and Q'. When a query is detached, there may or may not be such a variable common to both subqueries. The same detachment procedure is carried out repeatedly to each sub-query until each sub-query contains only one or two variables in the qualification part; in which case the sub-query/query is in the "non-reducible" form [Wong76].

(2) The *tuple substitution* step replaces a query involving two variables with a set of queries by replacing one of the variables with all its possible values. To illustrate this, the author again borrows another example

given by Wong [Wong76]. Consider the query:

Q: retrieve (S.Sname) where (Y.Snum=S.Snum)

Suppose that at this point Y.Snum is the projection:

Snum
101
107
203

Then, successive substitution of Y yields:

Q(101): retrieve (S.Sname) where (S.Snum=101)
Q(107): retrieve (S.Sname) where (S.Snum=107)
Q(203): retrieve (S.Sname) where (S.Snum=203)

(3) Any query that involves only one variable is handled by the **One Variable Query Processor** (the OVQP). The OVQP module is concerned solely with the efficient accessing of tuples from a single relation given a particular one-variable query. It therefore handles queries such as:

retrieve (S.Sname) where (S.Snum=101)
retrieve (P.Pnum) where (P.Pname='Bolt') and (P.Size=20)

The reason for using this algorithm is to restrict the size of the cross-product space needed to be searched [Jark84; Ston76; Wong76]. Effective execution of the above procedures is extremely crucial to the overall performance of system query processing.

2.3.5. Deferred Update and Utility Command Support

Process 4 in Figure 2.4 is comprised of two components: utility command supporting routines and deferred update routines.

As previously discussed, any update to the data-base is handled by first writing the tuples to be added, changed, or modified into a temporary file before any actual physical updating is done. Upon the completion of process **3**, the deferred update processor in process **4** becomes active, and performs the actual modifications requested and any updates to secondary indices which may be required as a final step in processing. Data integrity and the ease of control flow are among the main reasons for choosing the deferred update strategy [Ston76].

To explain how the deferred update procedure works, it is appropriate to quote Stonebraker [Ston76]:

> The deferred update file provides a log of updates to be made. Recovery is provided upon system crash by the RESTORE command. In this case the deferred update routine is requested to destroy the temporary file if it has not yet started processing it. If it has begun processing, it reprocesses the entired update file in such a way that the effect is the same as if it were processed exactly once from start to finish.
>
> Hence the update is "backed out" if deferred updating has not yet begun; otherwise it is processed to conclusion.

Stonebraker [Ston76] also points out that despite its conceptual and procedural simplicity, deferred update is a very expensive operation.

The utility command supporting routines are organized into several overlay programs. Required overlays are brought into the memory as needed. This is due to the fact that during the development of INGRES, there was not enough memory in the machine to hold all of the utility command supporting routines. Most of the utility supporting routines *update* or *read* the system relations by making AMI (Access Method Interface) calls. This section of the process is concerned with the support of commands such as CREATE, COPY, PRINT, etc. The contents of the utility supporting routines are otherwise trivial, and require no further elaboration.

CHAPTER 3

GENERAL DESIGN AND STRUCTURE OF FREDI

FREDI (**FR**ont **E**nd for **D**istributing the **I**NGRES system) is the front-end system developed for this research. FREDI can couple with a number of INGRES systems and function as a DDB system. It was developed on the network of four VAX-11/780's at the Department of Computer Science. Each VAX is running a 4.2 BSD UNIX operating system. The network is a ring-type network. INGRES is employed because it is a good data-base system and because of its availability on the UNIX operating system.

This chapter presents an overview to the design and the basic process structure of FREDI.

3.1. DESIGN OBJECTIVES

The first objective of this research is to build a DDB system that will be practical to use. In this regard we have to be concerned with how easily a user will adapt to the DDB system, as well as the performance of the DDB system. The second objective of this research is to investigate the following issues due to data distribution:

- data replication,
- site autonomy,
- site recovery,
- inter-site communication,
- query processing,
- concurrency control.

With respect to the second objective, the DDB system must be flexible, modular, and relatively easy to change so that it can be used as a testing ground.

Influenced by the above, the basic design of FREDI has five objectives:

- Ease of use.

  FREDI should provide the user with a "single system image". An INGRES user should be able to use FREDI right away without necessarily having an awareness of data distribution.

- Site autonomy.

  Each site should be self-sufficient. When a site fails, the only effect it should have on the DDB system is that the data in the failed site become inaccessible. The operation of other sites should not be otherwise affected.

- Response time.

  Most prototype DDB systems have a reputation of being slow; therefore it may be difficult to expect a "miracle" from FREDI. However, since FREDI is an interactive system, its response time should be at least within a tolerable limit.

- Site failure resilience.

  It is desirable that a DDB system should not be affected by site failures. FREDI should provide a degree of site failure tolerance. Data should continue to be available even after a site or some sites have failed.

- Modularity.

  FREDI should be highly modular. Old commands should be easily enhanced and new commands should be easily added. If desired, a new query processing strategy should be able to be employed without substantially changing the complete system.

3.2. A GENERAL DESCRIPTION OF FREDI

From the users' point of view, it might appear that a DDB system should support as many features as possible, and should give the users as much freedom as possible. However, in practice this may not be desirable. The more features a DDB system supports, and/or the more freedom a DDB system grants to the users, the more complex the DDB system becomes.

On the other hand, if a DDB system supports too few features and/or is too restrictive, the DDB system would not be very useful.

An important key to a good and viable DDB system therefore is to seek the right balance between the support of features and system complexity.

During the development of FREDI, many features and design alternatives were considered, and admittedly some of the decisions have been difficult. The direct result of these decisions is the external model of FREDI. It is summarized as follows:

- Relationship between FREDI and INGRES.

  FREDI acts as a front end to INGRES. When a user "logs-on" to FREDI, he "logs-on" to the particular FREDI DDB he specifies. In general, a FREDI DDB $\mathbf{X}$ is composed of the *local* INGRES data-bases $\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_n$, located at sites $\mathbf{S}_1, \mathbf{S}_1, \ldots, \mathbf{S}_n$, respectively,. each with the INGRES data-base name $\mathbf{X}$.

  In fact, the data-base system that FREDI couples with does not have to be a genuine INGRES system. With appropriate minor adjustments, FREDI can (at least theoretically) couple with any data-base system as long as the primary query language of the data-base system is QUEL.

All actual retrievals and updates to the data in the local data-bases are handled by INGRES; FREDI handles only issues regarding distribution.

- Choice of query language.

  The query language of FREDI is QUEL. This is consistent because the query language of INGRES is QUEL. Although other languages such as SQL [Date86] are also possible, adapting another query language would make FREDI more complicated to implement and may not be desirable.

- Granularity of data distribution.

  The granularity of data distribution in FREDI is a single relation. In a relational environment, this is an obvious logical choice to preserve simplicity.

- Data replication.

  Generally, there are two reasons for data replication: 1) performance, and 2) data availability/survivability. The support of data replication in FREDI is mainly for the latter reason. There are two reasons for this decision: 1) FREDI should be site failure resilient. 2) In a local area network environment, such as the one at the Department of Computer Science, the notion of a "near-by" site generally does not apply. Furthermore, it has been shown that to locate the *nearest* copy of the requested data in a multi-site environment is an NP-complete problem [Herv79, Yu84]; it is a complicated problem. The development of an algorithm that will identify the nearest site in a network will necessitate separate research.

To preserve simplicity, FREDI allows a maximum of only two copies for every relation. This provides single-site failure tolerance.

- Data fragmentation (data partition).

  Although data fragmentation is an interesting form of data distribution, FREDI does not support data fragmentation. This is because of the enormous complexity that can be introduced. As Lindsay [Lind85] points out, support of data fragmentation requires very serious design and implementation effort.

- Notion of primary copy.

  The notion of primary copy [Ston77; Ston80b; Date83] applies in FREDI. When a relation has two stored representations, one of the copies is designated as the primary copy, and the other as the secondary copy. Any data *update* is directed to the primary copy at the first instance. Data *retrieve* is directed to the *local* copy if there is one, otherwise it is directed to the primary copy. If one of the copies becomes unavailable, data reference is directed to the other copy (either primary or secondary) for both data update and data retrieval.

- Location and replication transparency.

  FREDI provides both location transparency and replication transparency.

  There are two reasons behind the support of these features: 1) One of our objectives is to provide the user with a "single system image". 2) A DDB system that provides location transparency and/or replication transparency is more flexible and more general.

  There is no need for fragmentation transparency because there is no data fragmentation in FREDI.

- Site autonomy and site recovery.

  It is always desirable that a DDB site should be highly autonomous. Each FREDI site is self-sufficient. The failure of one site does not interfere with the operations in other sites. When a "down" site recovers, the recovering site automatically reconciles the data in its local data-base and the data in other "up" sites.

  We shall assume only "clean" crashes (not the ones considered to be Byzantine failures[†]).

- Network partition.

  FREDI cannot cope with network partition. In any distributed system, it is impossible to distinguish a link failure from a remote site failure automatically. When a site cannot be reached, FREDI always assumes a site failure.

- Concurrency control and unit of a transaction.

  As a multi-user system, FREDI guarantees data consistency within a single user issued QUEL statement. In effect, one QUEL statement is defined as an **atomic unit** or a **transaction.** This is a consistent and obvious choice since one QUEL statement is defined as an atomic unit in INGRES. FREDI uses logical locks for concurrency control. Deadlock is prevented: the *processing* of an input command will not be executed unless the locks on all of the required resources can be acquired.

[†] A clean failure means a detectable failure. When it occurs, the site simply stops running. If a Byzantine failure occurs, the site continues to run after the crash, but performs incorrect actions [Bern84].

3.3. PROCESS STRUCTURE

A significant portion of FREDI is written in the programing language C. This is because C is the primary host programing language in UNIX. Some routines are constructed with the assistance of EQUEL. YACC is employed to assist the construction of parsing routines. LEX is used to assist the construction of the user interface — the interactive terminal monitor.

All inter-process communications between FREDI processes are provided by a 4.2 BSD UNIX facility call **socket.** The communicating processes may be on the same machine, or they may be on different machines. Sockets are also used to provide communication between FREDI and INGRES. Due to the data communication requirement, FREDI therefore requires each participating site to run a 4.2 BSD UNIX or compatible operating system.

The source listing for the complete prototype FREDI is over 10,500 lines. There are altogether seven (7) distinct FREDI processes. The names and the sizes of these processes are listed in Table 3.1.

Name of Process	**Size of Process** (in kilobytes)
interactive terminal monitor	41
input-analyzer/utility-processor	79
query-processor	68
three-way message relay process	32
network handler	59
slave process	46
lock-server	34

Table 3.1: Names and Sizes of all FREDI Processes

In addition to the seven processes, there is a utility process called **setupddb,** for setting up an INGRES data-base so that it can be used as a FREDI local data-base. The size of this process is about 31 kilobytes. This process is provided for convenience only.

FREDI operates under a master-slave model. We shall refer to the site where the query is originated as the **master site,** and each remote site as a **slave site.** In effect, the site where the user logs on is the master site. The master site takes full *control* of all distributed actions, while a slave site

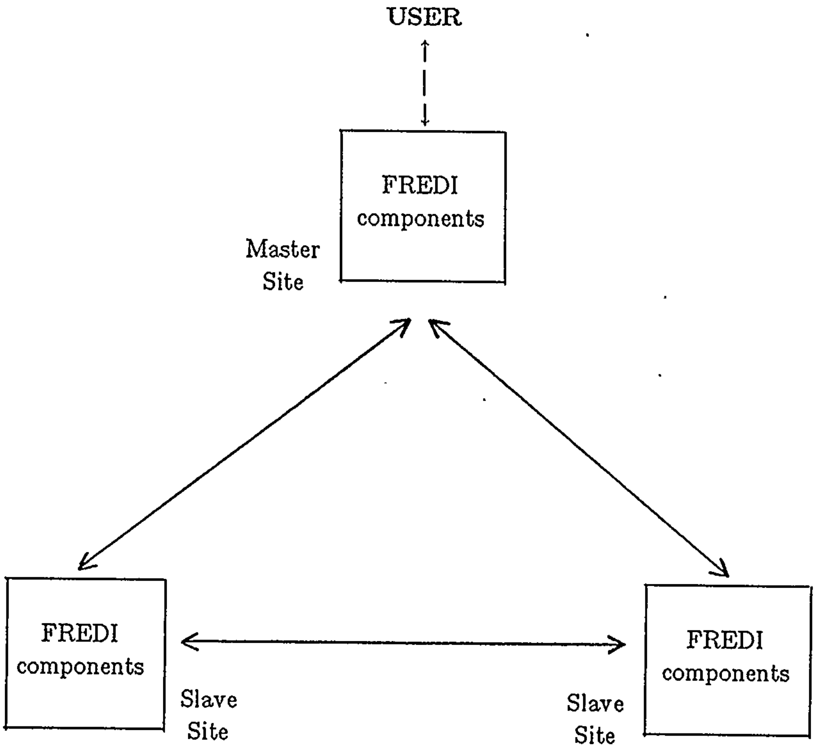


Figure 3.1: A Three Site Example of the Master-Slave Model.

merely acts passively as instructed by the master site. Compared to a model where all sites are equivalent, the master-slave model offers very high centralized control. This seems to be desirable because centralized control appears to be very straightforward.

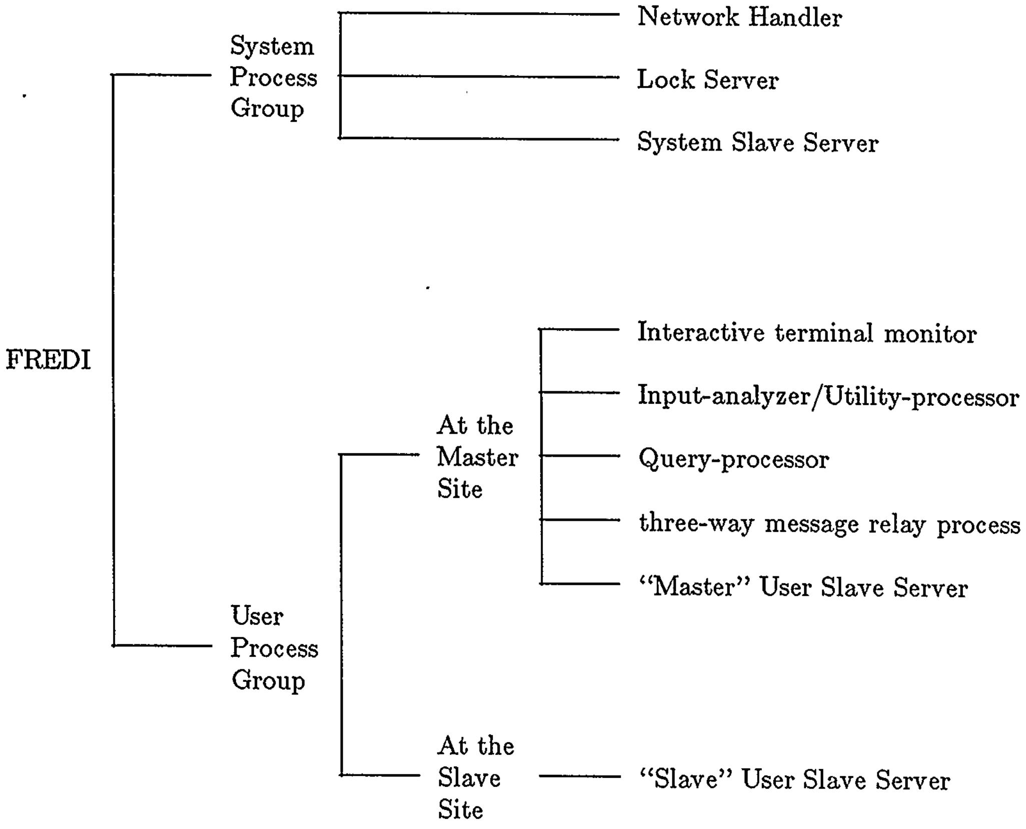

FREDI
System Process Group
Network Handler
Lock Server
System Slave Server
User Process Group
At the Master Site
Interactive terminal monitor
Input-analyzer/Utility-processor
Query-processor
three-way message relay process
"Master" User Slave Server
At the Slave Site
"Slave" User Slave Server

Figure 3.2: FREDI Process Hierarchy.

According to their roles and their volatility in memory, FREDI processes can be divided into two groups:

(1) **system process group** (created only at site start-up time), and
(2) **user process group** (created dynamically as needed).

For each DDB, a system process group remains active at each participating machine at *all times*, whereas an appropriate user process group is created at each site each time a user "enters" FREDI. All user process groups (one at each site) that *belong* to the user are destroyed when the user leaves FREDI. In other words, for each DDB at any time, each site has only one system process group but may have many user process groups — depending on the number of users that are using the DDB at the time.

3.3.1. The System Process Group

Each system process group is initiated at the site start-up time[†] at each site, and remains active until the site crashes.

A system process group is shared among users. Each system process group is composed of the following three processes:

(1) A **network handler.**
(2) A **system slave server.**
(3) A **lock server.**

The **network handler** is initiated first among the three processes. The lock server and the system slave server are initiated by a series of *fork* and *exec* operations, originated at the network handler.

[†] In order that FREDI be (re)started automatically whenever UNIX is rebooted, FREDI must be arranged as part of INGRES. This can be done, but requires special arrangements. As it stands, FREDI requires a manual (re)start.

The network handler contains four subsystems:

(1) The *communication subsystem* is responsible for relaying messages between local user process groups and remote network handlers.

(2) The *user slave server initiator* starts up a slave server process upon request.

(3) The *reliability subsystem* continuously monitors remote sites for failures. When a remote site failure is detected, it records this fact in the DDB

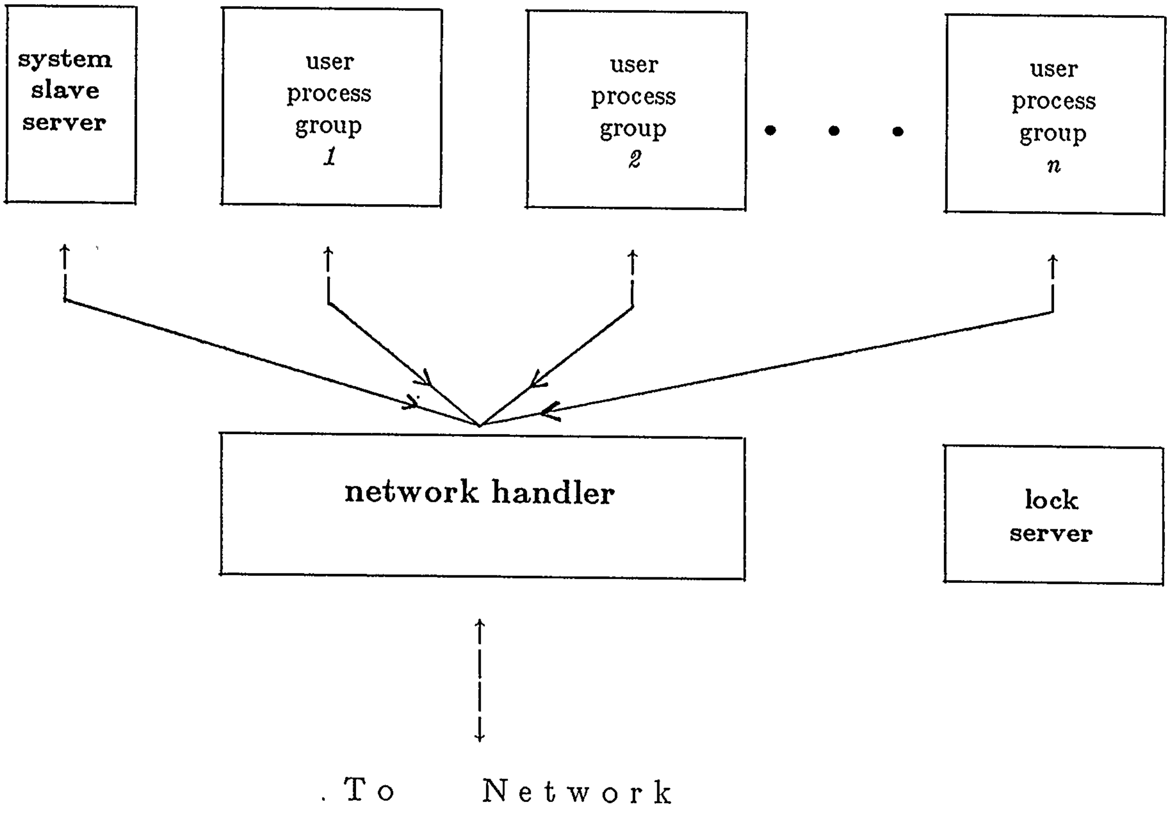


Figure 3.3: FREDI System Process Group Processes.

directory at the local site.

(4) The *recovery subsystem* is responsible for performing any recovery actions to the content of the local data-base during recovery time.

Since these subsystems have many functions in common, it makes sense to combine them in a single process.

The **system slave server** is a "passive" process. It is essential to the reconciliation of the content of the local data-base at a failed site and the local data-bases at other sites when the failed site recovers. It should be pointed out that a *system* slave server process and a *user* slave server process are identical processes.

The **lock server** is vital to the concurrency control of the data-base; in particular, it performs "test-and-set" operations — it *logically* locks and unlocks a relation on request. For each *local* data-base there is one lock server.

3.3.2. The User Process Group

User process groups can be sub-divided into two kinds:

(1) **master** user process group;
(2) **slave** user process group.

Every time a user enters FREDI, a master process group is created at the site where the user logs on (the master site). Corresponding to this master process group, a slave process group is created at *each active remote site* (the slave sites).

3.3.2.1. At the Master Site

Each master user process group contains the following collection of processes:

(1) An **interactive terminal monitor.**
(2) An **input-analyzer/utility-processor.**
(3) A **query-processor.**
(4) A **three-way message relay process.**
(5) A "**master**" **user slave server.**

The relationship between the processes within a master user process group is illustrated in Figure 3.4.

The interactive terminal monitor is the process first created when a new master process group is invoked. After the interactive monitor finishes the status check (existence of the DDB, locating the necessary system information, etc.), through a series of *fork* and *exec* operations, the rest of the processes are then created, one at a time.

The **interactive terminal monitor** is constructed with the aid of LEX. It buffers the user input in a temporary file, and allows the user to edit the content of the buffer.

The **input-analyzer/utility-processor** is constructed with the help of YACC. When there is user input arriving from the interactive terminal monitor, the input is analyzed for any syntactical errors. If an error is found, a message indicating this fact is sent back to the interactive terminal monitor and no further action occurs. In this case, the rest of the input in the input stream is also disregarded.

On the other hand, if a statement is considered syntactically correct, the statement is identified as a *utility command* (such as CREATE, PRINT, etc.) or

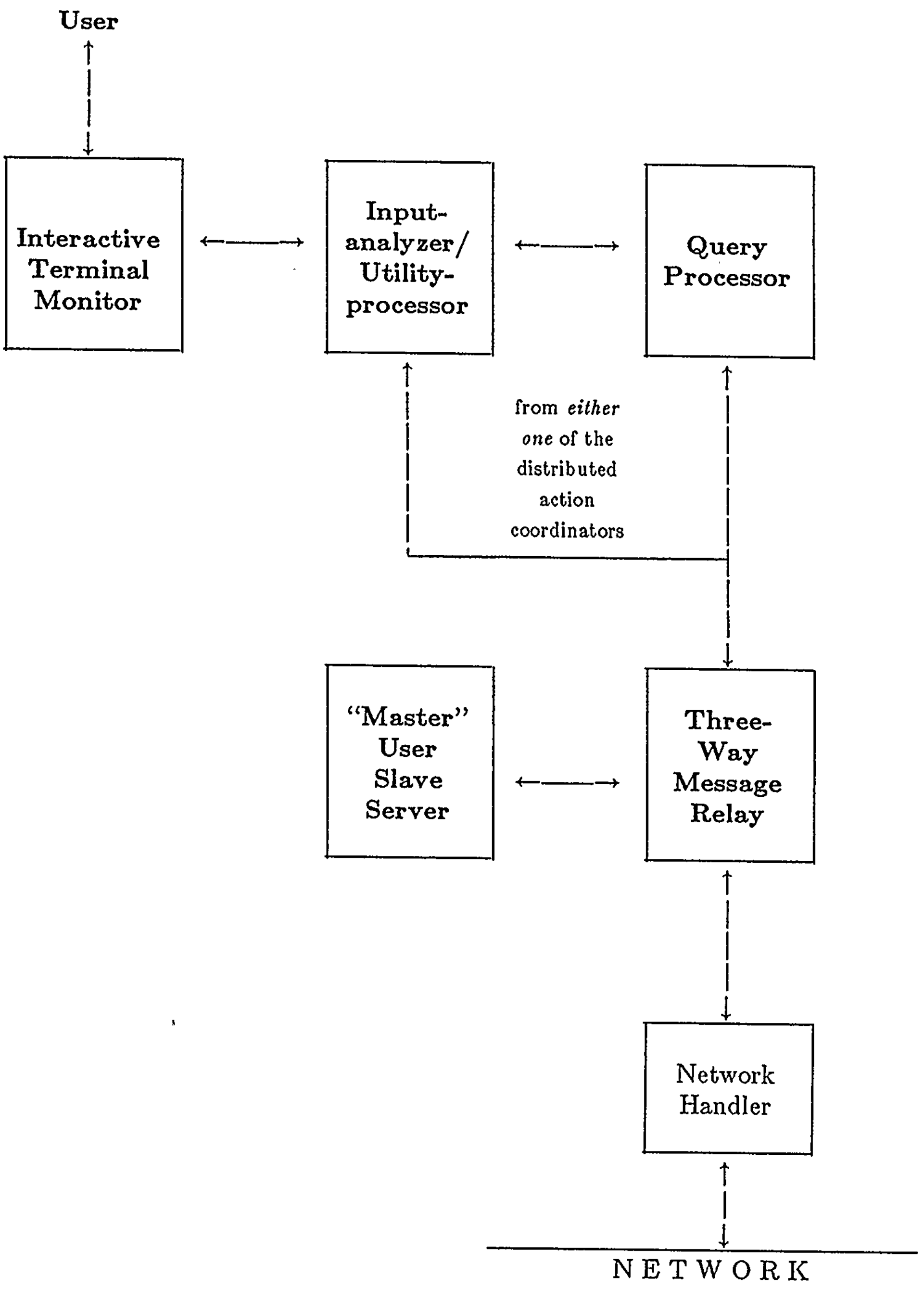

User
Interactive Terminal Monitor
Input-analyzer/ Utility-processor
Query Processor
from *either* *one* of the distributed action coordinators
"Master" User Slave Server
Three-Way Message Relay
Network Handler
NETWORK

Figure 3.4: User Process Group at Master Site.

a *query command* (either RETRIEVE, APPEND, DELETE, or REPLACE). If the statement is a query command, it is passed onto the query-processor for further processing. If the statement is a utility command, the appropriate utility routine(s) is invoked. The routine(s) checks for semantic correctness of the command and eventually performs the required actions if everything is correct.

The **query-processor** is also constructed with the aid of YACC. When a query command is passed down from the input-analyzer/utility-processor, the query-processor checks for the command's semantic correctness. If no error is found, the appropriate query processing routine(s) is invoked to perform whatever the query command requires.

The input-analyzer/utility-processor and the query-processor could have been built as a single process. In terms of execution efficiency this might be better since a query command would not have to be parsed twice. However, separating them into independent processes has one important advantage: since the query-processor is built independently, query processing strategy can be enhanced or even changed completely without any change in the rest of FREDI.

Because of their special roles in actually *coordinating* the distributed actions required, we shall indistinctly refer to both the input-analyzer/utility-processor and the query-processor as the **distributed action coordinator** in the rest of this thesis.

The existence of the **"master" user slave server** enables instructions issued by the distributed action coordinator to be executed locally as if they were instructions to be executed at a remote site. We shall explain what this means in further detail when we explain "Monolithic View of Sites" in section 3.6.

The **three-way message relay** process "relays" messages between the network handler, the master user slave server, and the distributed action coordinator (i.e. either the input-analyzer/utility-processor or the query-processor). It recognizes from which of the three processes a message comes when the message is intercepted. Using this information, and by "peeking" at the content of the message, the three-way message relay processor determines which one of the three processes should receive the message. The message is then routed to the appropriate process. The three-way message relay process is also part of the system that enables the distributed action coordinator to have a "monolithic view of sites".

3.3.2.2. At the Slave Site

At each slave site, the slave user process group constitutes only *one* process — the "**slave**" **user slave server.** Each one is dedicated to a *corresponding* master user process group.

3.3.3. Responsibilities of a Slave Server

The following are the services that a slave server provides (remember that the system slave server, the "master" user slave server, and the "slave" user slave server are identical processes):

(1) Submit one or more given QUEL statements to the local INGRES and run them. If instructed, the results from the INGRES execution of these QUEL statements are also forwarded back to the requestor.

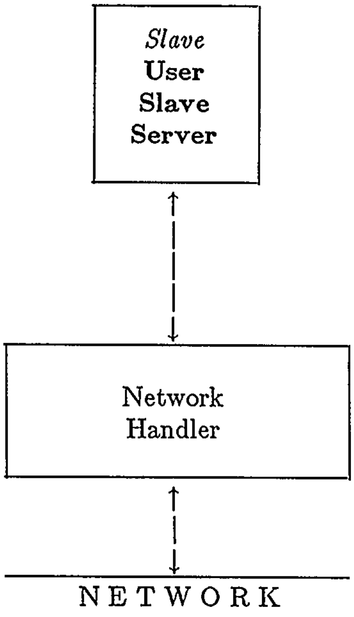


Figure 3.5: User Process Group at Slave Site.

(2) Receive the incoming (raw) data from the network and store the data in the *system temporary file* (a temporary file with a guaranteed unique file name).

(3) Send the content of the system temporary file to the slave server at a specified remote site.

(4) Copy the content of a specified file to the system temporary file, and vice versa.

(5) Invoke the local lock server to lock or unlock a specified local relation.

(6) Retrieve the names and the attributes of the domains of a specified local relation from the local INGRES data directory, and forward the result to the requestor.

The fact that a slave server is placed in every site and that a QUEL statement can be submitted to the local INGRES through a slave server is very useful. During the query processing time, all the master site (this implies the distributed action coordinator) needs to do is to formulate the appropriate QUEL statements, send them to the correct sites, and run them at the local INGRESes.

Since a slaver server knows what its responsibilities are, the messages passing to and from it can be expressed at a very high semantic level — instead of shipping the complete procedure which indicates *how* things should be done, only the instruction indicating *what* needs to be done is transmitted, such as: "run that QUEL statement", "lock that relation". Consequently, the sizes of the messages are generally very short.

3.3.4. Cost of Idle Processes

Although it is extremely likely that many of the processes in a FREDI system will remain idle most of the time, one should remember that an idle process does not contribute much cost. In a virtual memory operating system, such as UNIX, an idle process is likely to reside on disks.

3.4. USER INTERFACE

FREDI provides its own interactive terminal monitor for an user interface. However, there is no way for a user to provide his own C program as an interface to FREDI. If it were desirable to achieve such an objective, it could

Command	Abbreviations	Meaning
append	a	append input to the current workspace
clear	c	clear the terminal screen
ed	e	edit the current workspace using the user's *default* text editor
emacs	em	edit the current workspace using the UNIX text editor *emacs*
fred	f	edit the current workspace using the UNIX text editor *fred*
go	g	execute content of the current workspace
reset	r	clear the current workspace
shell	sh, s	escape to default shell
print	p	print content of the current workspace
write	w	copy the current workspace to a file

Table 3.2: A List of the Supported Monitor Commands

be accomplished by constructing a pre-compiler like that of EQUEL and provide a library of necessary C subroutines to accompany it.

FREDI's interactive terminal monitor behaves very similarly to the one provided by INGRES. This is desirable because a regular INGRES user will probably feel very comfortable with FREDI. Since some INGRES monitor commands are time consuming and/or difficult to implement, only the most often used subset of INGRES monitor commands are incorporated in FREDI. Hence, the following monitor commands are omitted: *list*, *eval*, *time*, *date*, *chdir*, *include*, *read*, *branch*, and *mark*. On the other hand, three new commands: **emacs**, **fred**, and **clear**, are added to make input editing easier in FREDI. It is interesting how easily these convenient features can be incorporated into the interactive terminal monitor. A list of the FREDI supported monitor commands is tabulated in Table 3.2.

3.5. QUERY PROCESSING STRATEGY

FREDI does not attempt to optimize the execution of a query. In fact, its query execution planning strategy is rather simple. It works as follows:

(1) Before any action is being carried out for a query, FREDI first determines the *target* site of the query — the site where FREDI thinks the *execution* of the query should actually take place.

(2) FREDI creates an *image* relation for each required relation that does not have a stored representation (either the primary or the secondary copy) in the target site. This involves shipping the contents of the required relations to the target site and making copies of these relations at the target site.

(3) The query is executed at the target site.

(4) If this is an update, and the relation to be updated is replicated, the secondary copy of the relation is updated accordingly. This is done either by sending all the changes of the relation to the secondary site, or by sending a complete copy of the relation to the secondary site.

(5) All the "image relations" are destroyed.

Although the query processing strategy used by FREDI is simple, it does have the advantage of handling complex queries rather easily because it takes advantage of all the query processing power of INGRES. Some QUEL features, such as aggregates, may be very difficult to support if the queries are handled in a truly distributed fashion. Furthermore, substantially more control messages would be required if queries were handled in a distributed fashion.

Unlike R^*, queries in FREDI are processed *within a single phase*, rather than separated into "compile" and "run-time" phases. It may be appropriate to say that queries in FREDI are interpreted.

In R^*, the basic idea is to get as much work done as possible during compile time, so as to reduce the amount of work needed to be done during run-time. This is feasible only because the majority of the SQL statements expected are to be expressed as embedded SQL statements in host language programs written in PL/1, COBOL, and assembly language. Perhaps for this reason, there is therefore no mention of the performance of R^* when SQL statements are issued interactively through QMF (Query Management facility).

Compared to R^*, the processing of queries in FREDI is handled in a very centralized manner. All verification of the existence and the availability of the resources required is carried out at the master site, with the only exception being the verification of the existence of a UNIX file at a remote site. During the time distributed actions are taking place, even though the distributed actions may be carried out at different remote sites, the master site still maintains a tight control on the *progress* of these actions. This fundamental manner of handling queries is desirable because very high site autonomy can be obtained this way. Admittedly, the master site may be considered to have lost its site autonomy momentarily during the period when distributed actions are carried out, since the execution at the master site depends on the execution at another site. However, this seems to be inevitable in any distributed system.

3.6. MONOLITHIC VIEW OF SITES

According to the FREDI architecture just described, for every "user" there is a slave server at every site (i.e. at both master and slave sites).

The existence of a slave server at every site, and the three-way message relay process at the master site, enables the distributed action coordinator to maintain a *monolithic view* of sites when it coordinates distributed actions. By "monolithic view" we mean that as far as the distributed action coordinator is concerned, *any site* is a remote site — including the local site (i.e. the

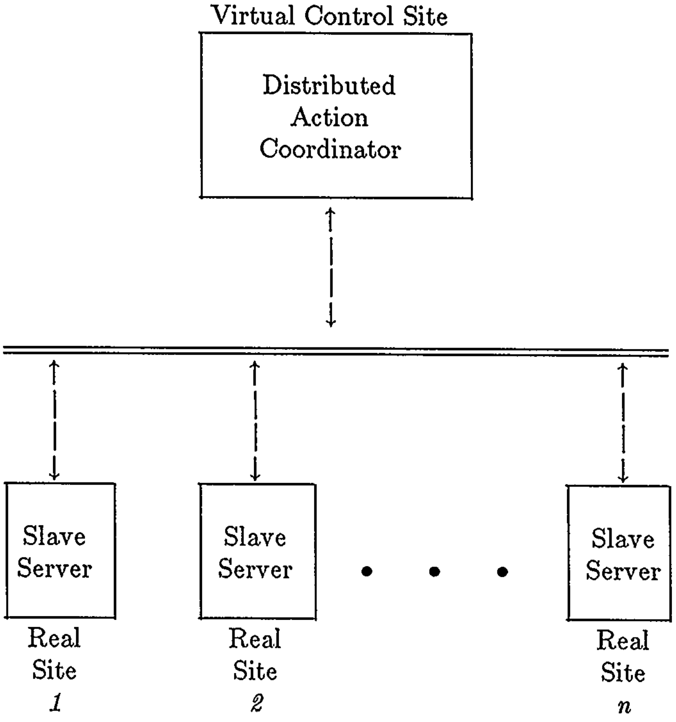


Figure 3.6: Monolithic View of Sites.

master site). A distributed operation is always considered to be carried out "remotely", whether the operation is actually being performed locally or at a genuine remote site.

The notion of *local site* is important during query optimization time. However, it is sometimes an undesirable complication to maintain a view that distinguishes a local operation from a remote operation during the time distributed actions are being carried out. If the "local site" notion is maintained, then every time the distributed action coordinator issues a distributed operation, it has to check if the operation is a local or a remote one. Any remote operation is forwarded to the slave server at the appropriate remote site, but a local operation must otherwise be taken care of by the distributed action coordinator itself. One can easily imagine the complications that can arise when a file is needed to be transported from site **A** to site **B**, when either site can be the local site, or when neither one is the local site.

In FREDI, the distributed action coordinator is logically being placed in a *virtual control site* when the distributed actions are being carried out. This is very handy. The distributed action coordinator simply issues "action instructions" to appropriate "remote" sites (including the master site), and monitors the progress of these distributed actions, without actually involving itself in a single action. Consequently, the basic procedure to carry out distributed actions is very simple: 1) Send an "instruction" to the site-*n* user slave server (can be master or slave), 2) Wait for the "complete" reply from the site-*n* user slave server, 3) Repeat step 1 and step 2 until all required distributed actions are completed. The virtual control site, of course, is only an imaginary site (i.e. an abstraction), and therefore no distributed action is needed to be performed there.

3.7. DATA DIRECTORY

FREDI's data directory contains information concerning the location or locations of where a relation is physically stored, the current availability status of the relation, and the names and the attributes of the domains of the relation.

FREDI's data directory is stored as INGRES user relations. This has two advantages. First, users may query the system data directory using the same query language that is used for other relations. Second, the data directory may be maintained by the same recovery and storage facilities as the rest of the data in the system.

The cost for storing the data directory as user relations is perhaps slower response time. Every reference to the FREDI data directory is an INGRES *query* operation; each takes about one second to complete.

The data directory is also fully replicated — each site maintains its own copy. Fully replicating the data directory at each site provides very high site autonomy, but at the expense of higher overhead during updates to the data directory. However, since in FREDI it is expected that the content of the data directory should be relatively stable, the site autonomy benefit seems to outweigh the extra cost required to update the data directory.

3.8. PARALLELISM

FREDI does not utilize parallelism during query processing time. The only time FREDI performs in parallel is when the copies of the data directory are updated at every site, such as in CREATE, BACKUP, or DESTROY operations.

The reasons why parallelism is not utilized during query processing time are:

(1) To be able to utilize any parallelism during query processing time, an immediate prerequisite is the availability of a very elaborate and sophisticated parser or analyzer that can analyze the input queries to figure out what to do. Preferably the parser is recursive and is able to generate the intermediate parallel action steps for the input queries recursively.

However, parsers generated with the assistance of YACC are non-recursive[†], and therefore using such a parser to generate the necessary intermediate parallel action steps for the input queries may be difficult. The following illustrates some of the difficulties: A RETRIEVE statement is sub-divided by the parser into two or more smaller RETRIEVE statements, which are to be executed in parallel. However, it is possible that some or all of these RETRIEVE statements generated can be sub-divided into finer granularities. To be able to figure this out, the same parser that parses the original RETRIEVE statement must be called for to parse each of the sub-divided RETRIEVE statements. In this case, recursion of the parser is required; an attribute that a parser generated with the aid of YACC does not possess.

Although a recursive-descent parser may be constructed using C, such a parser would be very complex and would require serious implementation effort.

[†] A parser generated with the aid of YACC uses LALR parsing method to parse input.

(2) The research community is still in search of a query processing strategy that will utilize parallelism well. The research on processing queries in parallel is still in its infancy. The problem of processing queries in parallel is an interesting but complex one, and the investigation of it should be done in separate research. Some pilot works in this area have been done by Apers, Hevner, and Yao [Aper83], Daniels *et al* [Dani82], Hevner and Yao [Hevn79], Yu and Chang [Yu84], and Yu *et al* [Yu85].

(3) It is much harder for a central control site to have a tight control on the progress of distributed operations if they are to be carried out in parallel at different sites.

3.9. MODIFICATIONS TO QUEL

Since QUEL was not originally designed for distributed environments, it is necessary to somewhat modify the original QUEL in order to adapt it to a distributed environment. Some of the original QUEL commands have to be altered semantically and/or syntactically, and some new ones have to be added. All the changes are upward compatible with the original QUEL: Query statements formatted according to the original syntax of QUEL are acceptable to FREDI. The design of these commands has been very carefully considered and the existence of each one is extremely vital to the usability of a DDB system. The following is a summary of all the changes (note: each optional parameter is enclosed by a pair of braces "{ }"):

- The modified syntax of the **copy** command:

copy relname (domname1 = format {, domname2 = format})
direction "filename" {**at** sitename}

With the "**at** sitename" parameter, it is possible to specify at which site the target file is/should be located. If this option is omitted, the master site (i.e. the site where the user is) is assumed. The location of the target file can be at any site, and has no relation to the location of where the relation is stored.

If the target file cannot be created or cannot be found, it is considered to be an execution error.

- The modified syntax of the **create** command:

```
create relname {at sitename}
               (domname1 = format {, domname2 = format})
```

Again, with the "**at** sitename" parameter, it is possible to specify at which site the new relation is to be situated. And again, if this parameter is missing, the master site is assumed. The name of the new relation must not already exist within the FREDI DDB. The created copy of the relation is considered to be the primary copy of the relation.

- The modified **destroy** command:

```
destroy relname
```

The **destroy** command destroys the specified relation regardless of whether the relation is replicated or not. The entries describing the relation in the FREDI data directory are also deleted. To simplify the complexity of the site recovery procedure, if a relation is replicated, the relation can be destroyed only when both of its stored representations are accessible.

- The modified **help** command:

  help {relname}

  This is a distributed extension of the INGRES **help** command. If the command is issued without a relation name, the information returned includes the names of all the relations in the FREDI DDB, and the locations of their stored representations (primary and secondary).

  If a particular relation is specified, then the information about that relation is listed. The information includes all of the relation's domains' names, the domains' attributes, and where the primary copy and the secondary copy of the relation are stored. If the specified relation does not exist in the FREDI DDB domain, it is considered to be an execution error.

- Syntax of the new command **move**:

  move relname {(primary-or-backup)} **to** sitename

  Since FREDI supports data replication, the "(primary-or-backup)" parameter specifies which copy of the relation (relname) is to be moved. If this optional parameter is omitted, the primary copy is assumed. The move must not result in a situation such that both the primary and the secondary copy of the relation will end up in the same site. It is a semantic error if a user tries to do so. The "to sitename" parameter specifies the destination.

- Syntax of the new command **backup:**

  backup relname {**at** sitename}

  The "at sitename" parameter specifies where the secondary copy of the relation should be located. If it is omitted, the master site is assumed. FREDI currently allows only one secondary copy for each relation. The **backup** operation must not result in both the primary copy and the secondary copy of the relation being in the same site. It is considered to be a semantic error if a user tries to do so.

- Syntax of the new command **unbackup:**

  unbackup relname

  This command removes the secondary copy of a relation. If the relation does not have a secondary copy, no action will be performed by FREDI, and it is not considered to be an error. To simplify site recovery procedures, for a replicated relation, this command can be executed only when both the site holding the primary copy and the site holding the secondary copy are connected and operational.

- Syntax of the new command **switch_primary:**

  switch_primary relname

  If a relation has two stored representations, then after this command the old primary copy will become the secondary copy, and the old secondary copy will become the new primary copy. There will be no effect if the relation does not have a secondary copy, and in such a case it is not considered to be an error in any way.

- The modified syntax of the **retrieve** command:

  retrieve {{**into**} relname {**at** sitename}} (target_list)
  {where qual}

  This is basically a distributed extension to the original **retrieve** command as defined in INGRES.

  If the "retrieve into" option is used, one can also optionally specify where (i.e. at which site) the result relation should be stored. If the "**at** sitename" parameter is omitted, the master site is assumed. The new relation must not already exist within the FREDI DDB.

- Syntax of the new command **upsite**

  upsite

  The response to this command is a list of the names of the sites that are currently connected and operational.

One should notice that the above commands do not require the user to know where a relation is stored or if it is replicated. Here we demonstrate how location transparency and replication transparency can be expressed in a query language.

3.10. USAGE RESTRICTIONS

If a FREDI DDB relation is replicated, FREDI cannot guarantee data integrity of the relation if a user tries to update any of the representations of the relation directly through INGRES. It is likely that the two copies of the relation will not be identical after the change.

FREDI maintains its own data directory in each local (INGRES) data-base as regular user relations. It is important that the integrity of this information be maintained. FREDI cannot function properly if such information is corrupted.

One also should not by-pass FREDI to create or destroy relations in a local data-base. Such a change cannot be reflected in FREDI's data directory. It is very important that FREDI's data directory has an accurate description of the current state of the DDB.

As the environment stands, one simple way to prevent the above from happening is to assign the owner of all the relations in each local INGRES data-base to be a single user, say, the FREDI system itself or a particular person such as the data-base administrator (DBA).

CHAPTER 4

INTER-PROCESS COMMUNICATION IN A DDB

General speaking, there are two kinds of inter-process communications: 1) communication between processes in the same machine, and 2) communication between processes in distinct machines.

Some DDB systems assume the existence of an inter-process communication environment. One example is R^*. Since R^* runs within a CICS environment, both local and remote inter-process communications are provided by CICS. An R^* process can send a message to any other R^* process through CICS. As far as R^* is concerned, the responsibility of inter-process communication is simply delegated to CICS. Once a message is handed over to CICS, the rest of the communication detail is of no concern to R^* *per se.*

However, in most situations a well established communication environment does not exist. It is more realistic for the DDB system to set up a communication environment of its own.

4.1. DESIGN CONSIDERATIONS

One should recognize that a DDB using a local area network (LAN) and a DDB using a long-haul network are very different. Usually, a long-haul network is more expensive and slower, but a LAN is less expensive and faster.

In order to make our discussion general enough but also descriptive of the communication model used by FREDI , we shall assume that when a user "logs on" to the DDB system, data in multiple sites are likely to be needed.

Secondly, we shall also assume that a connection established between process **A** and process **B**, and a connection established between process **B** and process **C**, does not imply that process **A** can communicate with process **C**. Communication connections are not generally *transitive*. Finally, we shall also assume all of the generic attributes of a LAN environment: the network at the Department of Computer Science is a typical LAN. These attributes include:

- Cost to transmit a byte of data from one site to any other site is relatively low compared to that of a long-haul network, say, ARPANET.
- Transmission rate is relatively high, in the order of 10 kilobytes per second.
- Propagation delay is negligible.
- Network is reliable. The network guarantees delivery of data (except when the destination site is down). The network also guarantees the data delivered are unrepeated and uncorrupted.
- Cost of an idle connection is very low.
- Broadcasting capability is not assumed. Although most LANs have broadcasting capability, the capability generally is not available to normal users; the network at the Department of Computer Science is an example.

Typically, in a very concise way, a communication session between two processes, **A** and **B**, can be described as the following: When process **A** wants to communicate with process **B**, process **A** *initiates* a connection. Process **A** goes into a *wait* state. When process **B** is ready, it *accept*s the request, and a *connection* (a virtual link, or a virtual circuit) is established. Process **A** proceeds to send data to process **B**, and vice versa. The connection may be *idle* intermittently during this period. When one or both of the processes decide that the connection is no longer needed, the connection is *disconnect*ed. In UNIX, a communication session between two processes follows the above pattern.

4.1.1. When Should an Inter-Site Connection Be Made?

An immediate question concerning inter-site connections in a DDB environment is: When should connections between DDB sites be made? Without going into any fine detail, three different approaches are suggested:

(1) A connection between two sites is made only when it is needed. Once the "transaction" is over, all connections are relinquished. We shall refer to this as the **ad-hoc** strategy.

(2) All the connections between all the sites are made in advance — before or at the time the user "logs on" on to the DDB system. We shall refer to this as the **full-connection** strategy.

(3) A mixture of 1) and 2). Connections are made at the user log-on time only between those sites whose names appear in a given per-user profile. Connections to other sites are made only when they are needed. Once the connections to a site are made, they may be retained until the user logs out, or they may be relinquished when they are considered to be no longer needed. The user may explicitly specify that the connections to a particular site be relinquished. Alternatively, the usefulness of the connections to a site may be determined by the length of time passed

since any of the connections were last used. We shall refer to this as the **prediction** strategy.

If remote access is frequent, the ad-hoc strategy is likely to suffer from poor response time. Establishing a connection is often time consuming and sometimes expensive; in UNIX, for example, it takes one or two seconds of elapsed time to establish a connection between two processes using sockets. Also, the control of *distributed actions* is likely to be very complicated since it is difficult to keep track of which connections were established. It is difficult for site **A** to learn about the connection status between site **B** and site **C** when the connection between them can be created and destroyed dynamically. Nevertheless, when the cost for *occupying* a link outweighs other factors, this approach may be appropriate.

The full-connection strategy is very straightforward, although a connection may lie idle most of the time.

The prediction strategy appears to be very flexible. Generally, a user should be able to predict in advance approximately the set of data he wants, and unnecessary connections may therefore be avoided. The cost of the flexibility is the increase in system complexity; for example, a "remote site" must be able to establish slave process(es) dynamically.

From the above evaluation, it seems that full-connection and prediction strategies are appropriate to a LAN environment. Even in a long-haul network such as ARPANET, where the cost depends only on the amount of data transmitted, full-connection and prediction strategies should also be considered. In FREDI, full connection strategy was chosen because it is straightforward to implement, and because FREDI operates in a LAN environment.

4.1.2. How Should Inter-Site Connections Be Shared?

In a multi-user distributed system, there are two obvious approaches to the sharing of communication resources among users:

(1) Each user has his own (conceptual) communication links between sites.

(2) All users share the same set of links between sites.

Ad-hoc and prediction connection strategies limit the option to choice 1), while full-connection strategy allows both choices.

Choice 1) has two potential drawbacks: a) It is possible that a huge number of connections will be needed. b) More seriously and detrimentally, it is possible that each user may have a different view as to the current status of the network.

In contrast, choice 2) needs only a fixed number of connections and provides each user with the same view to the current status of the network.

Another advantage of choice 2) is that it allows the DDB system to have more "system control" *at each site*. If all users are sharing the same set of site connections, it is simpler to implement procedures that will act upon the failure of a remote site, as well as site recovery procedures.

At first glance, one may suspect that it may cause a bottleneck if the communication link is shared by all DDB users. However, one should not worry about this because: 1) In most computer networks, a machine can only take in or send out a single stream of data through the network. More than one conceptual connection between two sites will not solve the traffic congestion problems, if physically there can only be a single stream of data passing between the two sites. 2) Even if a machine can genuinely send and receive multiple streams of data through the network in parallel (which is rare), the number of data streams will usually be very small, such as 2, 4, or 8 streams. Users soon still have to share the same set of physical data streams.

At the Department of Computer Science, the network is a ring-type network. It uses a token passing mechanism. Consequently, at any one time, only a single machine can send data through the network.

In light of the above, we conclude that in a DDB system it makes more sense for all users to share the same set of connections between DDB sites.

4.2. COMMUNICATION IMPLEMENTATION IN FREDI

FREDI uses a 4.2 BSD UNIX facility called **socket** to provide all interprocess communication: sockets are used to provide communication between FREDI processes at the same site and at different sites, and between a FREDI process and INGRES.

The other alternatives that can provide inter-process communication in 4.2 BSD UNIX are facilities called *pipe* and *pseudo-terminal.* However, sockets are chosen over these facilities because:

(1) These facilities cannot provide inter-site communication.

(2) A pipe connection can only provide uni-directional communication.

(3) There are only a limited number of pairs of pseudo-terminals available in each system; at the Department of Computer Science each machine has only 32 pairs. Pseudo-terminals are also used by many UNIX applications;

competition of resources may occur.

With respect to the FREDI process structure as described in chapter 3, there are three aspects of inter-process communication:

(1) Communication between all of the FREDI processes but not including the network handler.

(2) Communication between a (master or slave) user process group and a network handler.

(3) Communication between the network handlers on separate sites.

The first aspect is about *local* communication (i.e. communication between processes within the same site). The second and the third aspects are about *inter-site* communication, and are related to each other: they are concerned about how a *message*, when issued by one of the local user process groups, is sent to the user process group's counterpart in a specified remote site.

4.2.1. Socket: The Communication Facility Used

This section provides information about sockets.

In the 4.2 BSD UNIX, the concept of a **socket** is introduced to enable processes to communicate with each other; these processes may be on the same machine or on different machines. In previous versions of the UNIX system, the only system-provided function for inter-process communications was a **pipe**, and no readily available facility was provided for inter-machine communication. During those periods, without modifying the original UNIX operating system and adding some "local" facilities to the operating system, there was no way for a process on one machine to communicate with a process on another machine. For this reason, using just the facilities provided by UNIX, it was not be possible to construct a DDB system on a number of UNIX systems before the release of 4.2 BSD UNIX.

Whereas a pipe connection is only a *uni-directional* communication link, a socket connection is a *bi-directional* communication link. It is also interesting that in 4.2 BSD UNIX pipes are actually implemented internally as a pair of connected stream sockets [Leff83]. In effect, the socket, rather than the pipe, is the inter-process communication primitive in 4.2 BSD UNIX.

Employing UNIX terminology, it is said to be within the **UNIX domain** when sockets are used to provide communication between processes residing on the same machine. When sockets are used to provide communication between processes on separate machines, it is said to be within the **internet domain.** A UNIX domain socket may be identified by a UNIX directory name[†]. An internet domain socket may be identified by a "port number".

The communication model that FREDI employs falls into the *client/server* model [Leff83]. The general structures of a server process and a client process are illustrated in Pseudo-Code Listing 4.1 and 4.2, respectively.

It should be mentioned that on the server's side, when a "server socket" accepts a new "client connection", the "server socket" is *duplicated* — a new socket with the same attributes as the original one is created. Communication with the newly accepted client is through the new socket, and the original socket can be used to accept more clients. This way, a server can be the server of many clients at the same time. This is how the network handler in

---

[†] 4.2 BSD UNIX restricts the directory name to be a character string of no longer than 13 characters — the 13 character content plus a null byte.

```
main()
    {
        /* create a socket s using the socket primitive */
        /* s is the created socket */
        s = socket(information describing the socket nature);

        /* bind an address to the socket: */
        /*      bind an internet address if within internet domain */
        /*      bind a directory name if within UNIX domain */
        /* once bound, s is identified by the address */
        select_a_unique_address_or_directory();
        bind(s, the unique address/directory);

        /* indicate the willingness to accept connections */
        listen(maximum number of pending connections);


        for(;;)         /* s can be used to accept clients repeatedly */
            {
                /* accept a new connection */
                /* rendezvous occurs */
                /* socket s is duplicated */
                new_s = accept(s, information about the client)

                /* new_s is the connected socket to the client */

                /* communication to the client can begin (using new_s) */
                /* rendezvous occurs on all reads from a connected socket */
                                •
                more_process();
                                •
                                •
                                •
            }
    }
```

Pseudo-Code Listing 4.1: Typical Structure of a Server Process Using Sockets

FREDI can serve many user process groups at the same time.

UNIX treats a *connected* socket almost like a regular file descriptor, with one significant difference — in blocking mode (a UNIX attribute) rendezvous occurs on all *reads* on connected sockets. In other words, in blocking mode, if a process tries to *read* a connected socket that currently has nothing to be read, control of the process is suspended until there is something ready to be read. In contrast, when a process tries to *read* an opened file that has nothing to be read, the number zero (0) is returned to the call to indicate the end-of-

---

```
main()
    {
        /* create a socket using the socket primitive */
        s = socket();

        /* connect to the target server */
        /* server's address has to be known before hand */
        set up the server's address();
        connect(s, the server's address);

        /* transmission of data using the socket can begin */
        /* rendezvous occurs on all reads through the socket */
            •
        /* proceed to the rest of the program */
            •
        more_process();
            •
            •
            •
    }
```

Pseudo-Code Listing 4.2: Typical Structure of a Client Process Using Sockets

---

file condition, and execution of the process continues. This has two results: 1) Sockets can be used for process synchronization. FREDI takes advantage of this property and uses it for process synchronization. 2) There is no way to send a true end-of-file signal through a socket connection. In FREDI, the end-of-message signal is indicated by a sequence of special characters.

Because UNIX limits the number of file descriptors that a process can open, it also limits the number of clients that a server can serve. In 4.2 BSD UNIX, this number is set at 20. Consequently, according to the architecture of FREDI as described, this also limits the maximum number of users the a FREDI system can serve: each **network handler** in FREDI can serve only a limited number of user process groups. However, if it were desirable to serve a larger number of users, the following could be done: When the number of users served by a network handler reaches its limit, a new set of connections between sites are created, and a new network handler is created at each site. Half of the users are served by the new set of connections, and half of the users are still served by the old set of connections

4.2.2. Local Communication

The discussion of the communication between all FREDI processes but not including the network handler can be concentrated on three areas:

(1) Communication between processes within a master user process group.

(2) Communication between slave servers and a lock server.

(3) Communication between FREDI processes and INGRES.

Communication between the processes within a master user process group is provided by two UNIX domain sockets. One is between the

interactive terminal monitor and the **input-analyzer/utility-processor,** and the other is between the **input-analyzer/utility-process** and the **query-processor.** The names that bind to these sockets are randomly generated and each is guaranteed unique by measures taken within FREDI. Each socket is used to provide bi-directional communication as well as process synchronization.

A slave server can communicate to the local lock server by connecting to the lock server's UNIX domain socket. The lock server's socket is bound to a pre-selected name which is publicly known by other processes.

Two methods are used by FREDI processes to communicate with INGRES. In the first method , the EQUEL pre-compiler is utilized. It is used when a query can be pre-formulated and the result of the query is needed by the invoking FREDI process. Communication between a process and INGRES is truly established when the process "enters INGRES" (using **ingres dbname** command). At this time, pipes[†] are established between the invoking process and INGRES for the communication purpose.

In the second method, FREDI communicates with INGRES through a UNIX domain socket. In this case, queries are formulated under *ad-hoc* conditions and the content of the result from each query execution is of no concern to the execution of FREDI. Through the connected socket, INGRES actually thinks the input is coming interactively. This is desirable because commands to INGRES in this way can be formulated unrestrictedly.

[†] Actually, three pipes are created: One to pass data to INGRES, one to pass data from INGRES, and one to pass termination conditions from INGRES.

4.2.3. Inter-site Communication

Inter-site communication in FREDI is concerned with how a *message*, when issued by a user process group (in this context we imply either the distributed action coordinator or a slave server), is delivered to the *corresponding* user process group in a specified remote site.

The network handler plays a very important role in the inter-site

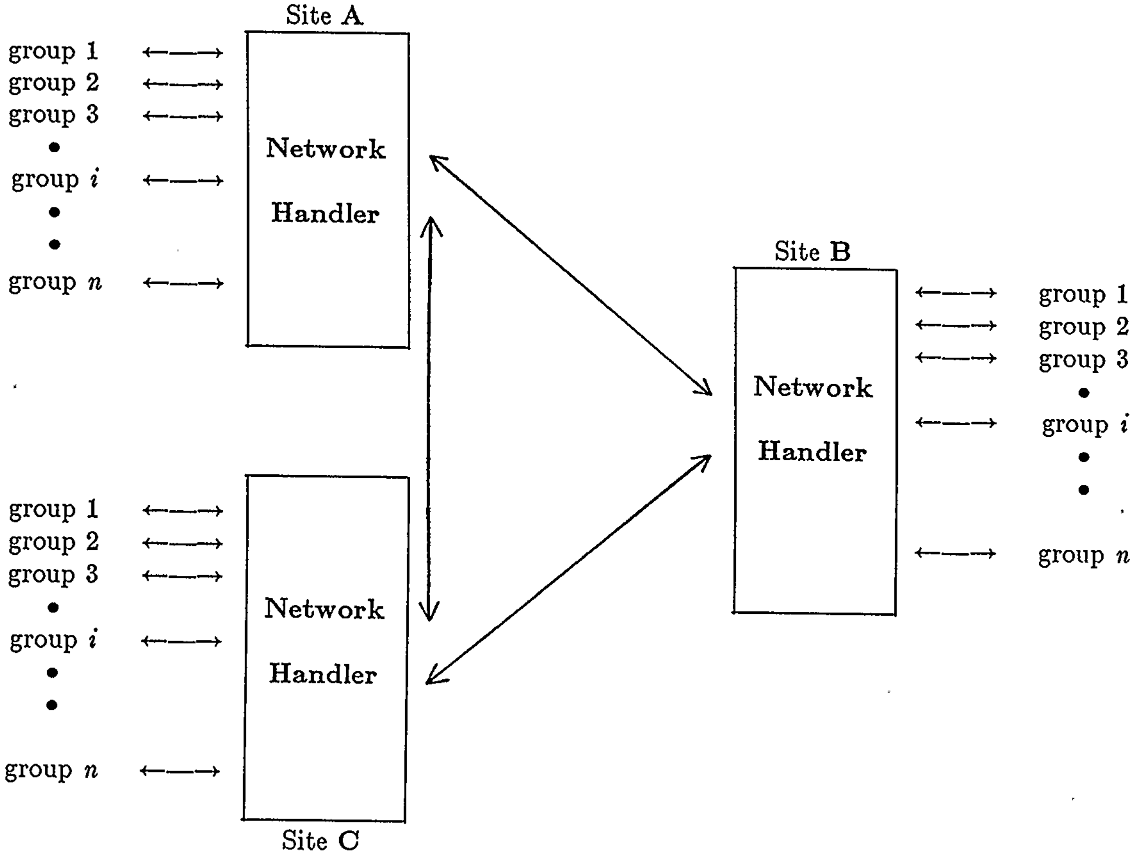


Figure 4.1: User Process to User Process Inter-site Communication.

communication aspect. In many ways, one may conceive of the network handler as a *telephone switch-board.* This concept is illustrated by Figure 4.1. When (user process) group *i* in site **A** wants to send a message to its counterpart (i.e. group *i*) in site **B**, it first forwards the message to the network handler in site **A**. The network handler in site **A** receives the message, identifies which group the message belongs to, tags this information onto the message, and forwards the message to the network handler in site **B**. Upon receiving the message, the network handler at site **B** identifies who (which local user process group) the message belongs to, and finds that it belongs to group *i*, using the tagged information. Removing the tagged information, the network handler forwards the message to group *i*. Group *i* receives the message.

Each network handler has two *permanent* sockets: an internet domain socket and a UNIX domain socket. The purpose of the internet domain socket is to accept connections from network handlers at other sites. The port number of the internet domain socket of each network handler (one at each site) must be unique across the network. These port numbers[†] must be chosen manually in advance during "system configuration" time. The purpose of the UNIX domain socket is to accept connections from newly created user process groups. Every time a new master user process group is created, the **three-way message relay process** of the group *connects* to the local network handler. The name of the UNIX socket needs to be unique only within the site. It also must be chosen manually in advance during "system configuration" time.

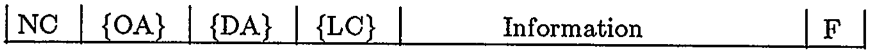
[†] In the 4.2 BSD UNIX those ports numbered 0 through 1023 are reserved for UNIX applications (such as *rlogin, rwho, ruptime,* etc.).

In order to shorten the length and to increase the semantic level of the messages that flow between a user process group and network handler, and between two network handlers situated in different sites, it is necessary to format the messages in a fixed way.

Due to the difficulties in choosing appropriate and descriptive names, we shall (inaccurately and arbitrarily) refer to the *format* of the messages that are passed between a user process group and a network handler as **local-phase message format**, and the *format* of the messages that are passed between network handlers as **remote-phase message format.**

4.2.3.1. Local-Phase Message Format

The **local-phase message format** is a set of rules that govern the flow of data between a user process group and a network handler. We shall



NC = Network Control
OA = Origin Address
DA = Destination Address
LC = Local Control
F = Flag

Note: A pair of braces ("{}") indicates an optional item.

Figure 4.2: Local-Phase Message Format.

(arbitrarily) use the term **block** to refer to the vehicle for carrying messages between a user process group and a network handler. Because data transmission through sockets is *byte oriented*, it may therefore be appropriate to say that a block is also byte oriented. The various fields that comprise a block are as follows:

NC	Network Control
OA	Origin Address
DA	Destination Address
LC	Local Control
I	Information
F	Flag

The **network control** (NC) field is a one byte field. It identifies the *type* of message the block contains. The network control field can be:

macro symbol	**meaning**
B_USER_MSG	user message
B_SITE_RQST	remote site survival information request
B_SITE_LIST	remote site survival information reply
B_DISCONN	disconnect the socket connection

For a user message (B_USER_MSG), the network handler processes the block as follows: If the block just arrived from a user process group, the network handler assembles the message as part of a **frame**, and sends the frame to the appropriate remote site (i.e. the network handler at the remote site). If the block was just received as part of a frame delivered from a remote site, the network handler sends the complete block to the appropriate user process group according to the information that is attached as part of the frame received. We shall discuss frames shortly.

When a user process group sends a block to the network handler indicating that the survival information (B_SITE_RQST) about remote sites is needed, the network handler replies with a block (B_SITE_LIST) containing a list of all of the currently operational sites. The network handler maintains the knowledge on the survival of all the remote sites.

If the network handler receives the disconnect request (B_DISCONN), it *closes* down the socket connection connected to the user process group in question.

The origin address (OA) field, the destination address (DA) field, and the local control (LC) field are present in a block only when the network control (NC) field identifies the block as a user message.

The **origin address** field identifies the site from which the block is originated.

The **destination address** field identifies the site to which the block is heading.

The **local control** field identifies to the recipient user process group *what type* of user message the block contains. It can be one of:

macro symbol	**meaning**
BL_QUEL	QUEL statement(s)
BL_QUELR	QUEL statement(s), result of the execution wanted
BL_RATTR	attributes of the domains of a local relation wanted
BL_CMD	special command, such as to lock and unlock a local relation, to read data from a file, etc.
BL_DATA	raw data
BL_REPLY	done reply (from slave server).

These are "instructional messages" from one slave server to another slave server, from a distributed action to a slave server, or vice versa. The first four can only be issued by a distributed coordinator (i.e. either the **input-analyzer/utility-processor** or the **query-processor**). BL_DATA can only be issued by a slave server to another slave server, and BL_REPLY can be issued by a slave server to the distributed coordinator. These messages correspond to the responsibilities performed by a slave server. We have discussed the responsibilities of a slave server in chapter 3.

The **flag** field indicates the *end-of-block*. It is a two byte field, and is (arbitrarily) designated to be two consecutive "]" characters (i.e.]]). This is necessary because a socket cannot deliver an end-of-file signal. In order that a "]" character can be delivered as data in the I-field (information field), the *byte stuffing* technique is used [Hous79; Tech80]. Every "]" character is represented as the "][" pattern in the I-field.

4.2.3.2. Network-Phase Message Format

The **network-phase message format** is a set of rules that govern the flow of data between network handlers. We shall (again, arbitrarily) refer to the vehicle for carrying messages between network handlers as a **frame.** A frame is also *byte oriented.* It has the general format as illustrated in Figure 4.3.

C	{PL}	Information	F

C = Control
PL = Process Label
F = Flag

Figure 4.3: Network-Phase Frame Format.

The various fields that comprise the frame are as follows:

C	Control
PL	Process Label
I	Information
F	Flag

The **control** (C) field is a one byte field. It specifies what *type* of message the frame contains. The control field can be:

macro symbol	**meaning**
F_USER_MSG	user message
F_CREAT_SLAV	create a new slave server
F_RU_THERE	"are you there" enquiry
F_IM_HERE	"I am here" reply

If the control field indicates F_USER_MSG, the information field contains one **block** of user information.

If the control field indicates F_CREATE_SLAV, the recipient network handler is requested to create a new slave server at that site.

If a network handler receives a F_RU_THERE request, it replies with a frame indicating F_IM_HERE, to indicate the site's survival.

The **flag** identifies the *end of frame*. It is a two-byte field. The flag is (once again, arbitrary) chosen to be two consecutive "}" characters (i.e. }}). A "}" character in the I-field is represented as "}{".

Note that no synchronization flag is necessary. The appearance of a flag automatically implies the beginning of the next frame.

4.2.4. Detection of a Failed Site

On some networks, detection of a site failure is provided as an internal feature (e.g. the early ARPANET NCP protocol and SNA's virtual circuit protocol [Bern84]). Others do not (e.g., the current ARPANET TCP/IP protocol [Bern84]). At the network at the Department of Computer Science, such as feature is not provided by the network.

At the network at the Department of Computer Science, sometimes when the remote end of an internet domain socket connection is disconnected (because of the other end of the connection being *close*d, the remote process crashing, or even the remote site crashing), the following may be observed at the local end: The SELECT system call in UNIX will report that the socket in question has some data ready to be read, but the READ system call will report the socket has nothing (i.e. zero byte) ready to be read. In this case, the read also will not suspend the control of the process (remember that rendezvous normally occurs on a *read* to a socket). However, it should be pointed out again that the above response is not guaranteed to happen *every time* an internet domain socket connection is disconnected at the remote end.

Although FREDI takes advantage of the above special behavior of UNIX to detect failure of a remote site, a high level time out is also needed to remedy the fact that the behavior does not occur consistently. High level time out is not satisfactory, but is workable.

Time out in a distributed environment is always considered a very difficult thing to do. It is often very hard to estimate the length of time a remote site needs to issue a reply message. In FREDI, this problem is alleviated slightly because inter-site messages are sent from the network handler at the source site to the network handler at the destination site. Since no network handler is responsible for any kind of computational purpose, there is no need to worry that a network handler would be too busy to reply to an "are you alive" enquiry message. In this respect, one may find it reasonable to expect a reply to be responded within a relatively short period of time, such as within a few seconds.

CHAPTER 5

PERFORMANCE OF FREDI

5.1. RESPONSE TIME OF FREDI

One of the major objectives of FREDI is *reasonable response time.* Table 5.1 shows some sample results of FREDI's response time. These results are included here only to provide an indication of FREDI's performance, and should not be used as benchmarks.

Whether the performance of a DDB system is acceptable or not clearly depends on the nature of the applications involved; in many non-critical applications using small to medium databases (e.g. relations with less than 4,000 tuples), response times of less than 60 seconds may be acceptable. Using these admittedly vague criteria and the numbers as shown in table 5.1 as a guideline, the performance of FREDI can be considered to be acceptable — each command was completed within seconds, rather than minutes or even hours. The results in table 5.1 were obtained while the machines and the network were under light load. The load average on each machine did not exceed 5 — there were less than 5 jobs on the job queue in the operating system. The network data transmission rate was about 15 to 25 kilobytes per second. The same command was run three times and the real elapsed time of each run was taken. The median of the three figures obtained for each command was included in the table. It was noted that the deviation of the three figures obtained was no more than four seconds in all cases. For this reason, the median of each of the three figures in each case would have been very close, if not identical, to the average of the figures.

In general, for "single-site operations" FREDI offers performance comparable to that of INGRES. By "single-site operations" we mean operations that do not require shipping of any relations from one site to another. Furthermore, there seems to be no noticeable difference in response time between *remote* "single-site operations" and *local* "single-site

Command	**Real Elapsed Time** (in seconds)	**Remarks**
print	< 1.0	• local relation • almost instantaneous
print	5.0	• remote relation • small relation, less than 50 tuples
print	36.0	• remote relation • large relation, 4,000 tuples, each tuple is 20 bytes wide
print	27	• remote relation • large relation, 2,000 tuples, each tuple is 20 bytes wide
move	6.0	• small relation, less than 50 tuples
move	35.0	• large relation, 4,000 tuples, each tuple is 20 bytes wide
move	25.0	• large relation, 2,000 tuples, each tuple is 20 bytes wide
retrieve	19.0	• multi-site retrieve • 3 relations shipped • small relations, each less than 50 tuples.
retrieve	36.0	• multi-site retrieve • 1 relation shipped • large relation, 4,000 tuples, each tuple is 20 bytes wide
retrieve	21.0	• multi-site retrieve • 1 relation shipped • large relation, 2,000 tuples, each tuple is 20 bytes wide

Table 5.1: Some Sample Figures of FREDI's Response Time

operations". This is probably because the control messages involved in a remote "single-site operations" are so short that they do not contribute any significant overhead.

For "multi-site operations", when the sizes of the relations involved are small, such as 50 tuples each, shipping of one extra relation seems to add roughly another 4 to 6 seconds to the response time.

When the relations involved are large, such as in the order of 1,000 tuples each, the sizes of the relations have a significant impact on the final response time. It is, however, difficult to measure exactly *how* the sizes of the relations affect the response time, since both the load of the machines and the load of the network are important factors which fluctuate from time to time. Hence, the length of time it takes to execute one command on one occasion and the length of time it takes to execute the same command using the same data and the same machines on another occasion can be very much different.

In spite of the dependency on the network load and the machine load, to give an estimate how different sizes of a relation affect the response time, the following was observed: In one test, execution of a command that needed to ship an 80,000 byte relation across sites took about 36 seconds to complete. While the machines and the network were under almost the same load, the same command took about 21 seconds to complete, when a 40,000 bytes relation was shipped. During the test, the network was offering around 20 kilobytes per second of transmission rate, and the load average on each machine was about 2.5.

5.2. SOURCES OF FREDI OVERHEAD

There are three factors that could affect FREDI's performance (in terms of response time):

(1) The overhead contributed directly by FREDI processes.

(2) The network overhead — the time it takes to ship the required data across sites through the network.

(3) The overhead it takes to execute a single QUEL statement in INGRES.

It is believed that the overhead generated directly by FREDI processes is small. The fact that for "single-site operations" FREDI offers a performance comparable to that of INGRES may prove this point.

From observation, it has been found that generally the network at the Department of Computer Science offers a data transmission rate ranging from 10 kilobytes to 30 kilobytes, according to the current traffic load of the network. Occasionally, the network can even offer a data transmission rate of well over 30 kilobytes. This is typical in a local area network. Based on these observations, we can conclude that when the amount of data shipped across sites is less than in the order of 10 kilobytes, the overhead due to network usage would be very small. When the amount of data involved is in the order of 10 kilobytes, one might expect every 20 kilobytes of data will add an average of about one second to the response time. Furthermore, since the control messages required by FREDI are generally short — about 30 bytes to 50 bytes each, the network overhead due to control messages is almost negligible.

The third factor, the overhead directly from INGRES, seems to have a lot of effect on the performance of FREDI. The effect of INGRES performance to FREDI "single-site operations" is obvious and needs no further explanation. To explain how INGRES affects FREDI "multi-site operations", it is appropriate to explain how a relation is shipped from one site to another site. In FREDI, the shipping of a relation from site **A** to site **B** involves the following steps:

(1) Site **A**: Issue an appropriate COPY INTO command to INGRES, and copy the target relation into a temporary file.

(2) Site **A**: Send the content of the temporary file to site **B** through the network.

(3) Site **B**: Receive the data from the network, and store the data in a temporary file (not to be confused with the temporary file at site **A**).

(4) Master Site (can be site **A**, site **B**, or a third site **C**): Issue an appropriate RETRIEVE command to INGRES, and interrogate the FREDI data directory to find out the descriptive information (such as the names of the domains, types of domains, etc.) about the target relation.

(5) Site **B**: Issue an appropriate CREATE command to INGRES, and create the new relation.

(6) Site **B**: Issue an appropriate COPY FROM command, and copy the content of the temporary file into the newly created relation.

As one may see, to make an image of a relation at another site actually involves four INGRES QUEL statements: a COPY INTO, a simple RETRIEVE, a CREATE, and a COPY FROM. From observation, when the machine is under light load and the amount of data involved is very little (such as a 50 tuple relation), it takes about one to two seconds to execute each of the above commands in INGRES. The length of time increases as the amount of data involved increases. One can quickly see that to make an image of a small relation at another site could take as much as 6 seconds of INGRES execution time, and longer or much longer for a larger relation. The *shipping* of the data itself therefore may not be very time consuming *per se*, but the four INGRES operations that constitute the shipping procedure contribute significantly to the delay in response time. Referring to table 5.1 again, it makes sense that the multi-site RETRIEVE command that involves shipping of three small relations takes about 19 seconds to execute.

5.3. SUGGESTED WAYS TO IMPROVE PERFORMANCE

Three ways are suggested that may improve FREDI's performance:

(1) Query optimization.

FREDI does not attempt to optimize the way it handles queries. FREDI processes *all* queries using the same method: Ship all the required relations to the "target" site, and perform all the necessary "computation" there.

It has been shown that when the relations in the DDB are not very big (such as less than 50 tuples each), the main contributor of overhead in FREDI is the *number* of relations that are needed to be shipped across sites. Using this basic idea as a guideline, an immediate improved version of the above query processing strategy would be: 1) Select the site X where it is storing the majority of the number of the required relations. 2) Ship all other required relations to site X. 3) Perform the necessary "computation" at site X. 4) Ship the result of the "computation" from site X to the "target" site, and perform any final steps that may be required. If site X is the same as the "target" site, this final step may be omitted.

When the sizes of the relations in the DDB are large, the local processing overhead as well as the network usage overhead become significant. Query optimization in this case should be aimed at two directions: 1) Restricting the *amount* of data shipped across sites. 2) Taking advantage of parallelism. These two aspects are related to each other. To increase parallelism in most cases means that more data are needed to be shipped across sites. At this stage, the research community is still in search of a good algorithm that will restrict the amount of data shipped across sites and utilize parallelism well at the same time. Furthermore, one must realize that to derive such a query processing strategy requires very serious effort.

(2) Storage of data directories.

In FREDI, data directories are stored as user relations in INGRES. As pointed out, a simple RETRIEVE command in INGRES sometimes takes almost over a second to execute. Since data directories are being interrogated very often by FREDI, FREDI response time would probably be improved if FREDI were to store its data directories as UNIX files. However, storing the data directory as user relations still seems to be easier to manage and also more general and flexible.

(3) INGRES capability.

FREDI actually does not require all of the facilities provided by INGRES. A smaller version of INGRES that contains only what is required by FREDI would probably be faster than the full version of INGRES. If FREDI were coupled with this subset INGRES, it might improve somewhat the performance of FREDI.

CHAPTER 6

DATA DISTRIBUTION IN A DDB

The way data and data directories are distributed in a DDB system has a very serious and direct effect on the complexity of the DDB. In this chapter, we shall discuss how different forms of data distribution affect the complexity of a DDB system.

6.1. DATA DIRECTORY IN A DDB SYSTEM

The **data directory** (or simply **directory**), sometimes referred to as **data dictionary**, or **system catalog**, contains information describing how user data are stored in the DDB. In general, the information stored includes the location or locations where a relation is physically stored, its cardinality, and the names of domains. For each domain, further information is also provided such as data type, high and low domain value, and domain cardinality. In addition, sometimes information such as security controls, data integrity controls, and statistical information is also included [Chu79, Brac80].

Data directory systems can be classified into the following basic categories [Chu79]:

(1) Centralized.

Directory information is stored in only one of the DDB nodes.

(2) Multiple Masters.

When DDB nodes are clustered into groups (i.e. star network), it is sometimes effective to provide a master directory to serve each cluster.

(3) Localized (or Fully Partitioned).

Each node contains directory information for only the data stored at that node.

A major drawback of such a directory system is that it has high communication cost and requires high search time for non-local data. For a DDB of n nodes, it requires an average of $(n - 1) / 2$ directory queries to locate the information about non-local data. However, if each node contains a routing table which routes the query to another node rather than returning the negative query reply to the sender, the expected total communication cost may be greatly reduced (by a factor of ξ, $0 < \xi < 1$), particularly if the routing table takes into consideration the probability of finding the data in the the directory.

(4) Distributed (or Fully Replicated).

Each node contains the complete information about the DDB.

6.2. FORMS OF DATA DISTRIBUTION

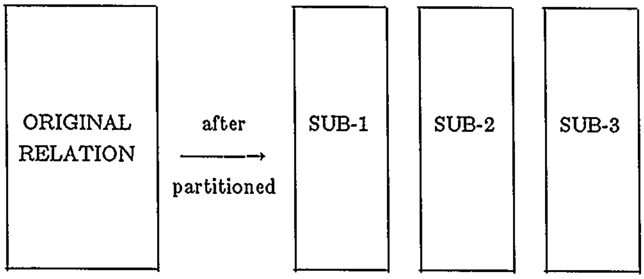
In relational environments, data can be distributed in one of the following forms:

(1) replication,
(2) horizontal partition,
(3) vertical partition,
(4) a mixture of any of the above,
(5) no replication, no horizontal partition, and no vertical partition.

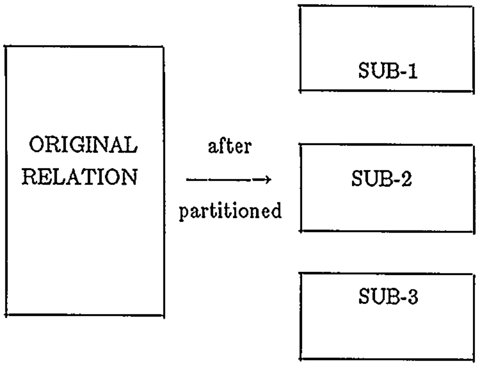
Replication means that copies of the same data are stored at different sites. Updates are propagated to all replicas synchronously. Replicated data can be used to provide higher availability and/or faster response for read operations, but at the expense of requiring more overhead for updates.

Vertical partition means storing a relation with different projections of the relation at different sites. Correlation domains are maintained at each site to allow the complete relation to be reconstructed.

Horizontal partition refers to partitioning a relation and storing each portion of the relation in a different site according to distribution criteria



6.1A: Vertical partition



6.1B: Horizontal partition

Figure 6.1: Vertical and Horizontal Partitions of a Relation.

based on the values of specified domains. Updates to a partitioned relation may therefore cause tuples to move from one site to another.

The main purpose of both vertical partition and horizontal partition is to store the data close to where they are most often used, and thus (hopefully) lessen the overall access overhead to these data. Figure 6.1 illustrates both vertical and horizontal partitions.

In practice, to support data distribution at a granularity finer than a relation is problematic. Furthermore, to support more than one of replication, horizontal partition and vertical partition will likely increase the system complexity astronomically.

6.3. DATA DISTRIBUTION IN FREDI

FREDI supports data replication only, for the following reasons:

(1) Among the three alternatives of replication, horizontal partition and vertical partition, replication seems generically to be more important and is also more appropriate in a local-area network environment such as the one at the Department of Computer Science. Date [Date83] points out that a "true" DDB system should support at least some degree of data replication. In a local area network environment, the network usually has a high data transmission rate and is very reliable. In contrast, some of the machines occasionally do go down for a variety of reasons. It is therefore a good environment for illustrating the data availability problem.

(2) Both horizontal partition and vertical partition require the granularity of data distribution to be finer than a relation, which would make the data

directory extremely difficult to design and manage. In addition, query processing in this case would be very complex.

In a FREDI DDB, the unit of data distribution is a relation. Each relation can have up to two copies of data representation. The notion of primary copy applies: the first copy is treated as the *primary* copy, and the other optional copy as the *secondary* copy. The two copies must be stored in separate sites.

If a relation has only one data representation, then any accesses and updates to the relation obviously have to refer to that copy. If that copy is not available, the relation is not available.

If a relation has two copies of representations, unless there is a copy of the relation locally, data references are always made to the primary copy if both copies are available. All updates to the relation are made to the primary copy first, and then to the secondary copy.

The obvious way to handle site failures is to ignore failed sites [Bern84]. When a site holding one of the two copies is not available, the system applies the update to the available copy and ignores the copy at the "down" site.

6.4. FREDI'S DATA DIRECTORY SYSTEM

FREDI's data directory consists of two INGRES relations. It is fully replicated.

Storing the data directory as INGRES relations has the following advantages:

(1) The organization of the directory can be easily enhanced if such a need arises.

(2) During site recovery time, an up-to-date data directory can be shipped in from another site as if the data directory were user relations. No special recovery data transfer step is required.

(3) Users can examine underlying contents of the data directory if they want to.

The drawbacks of storing the directory as INGRES relations are:

(1) Since a RETRIEVE to an INGRES relation is relatively slow (about one second), the frequent references to the data directory can slow down the performance of FREDI.

(2) The data directory may be exposed to more security problems.

The directory is fully replicated for the following reasons:

(1) To avoid the failure of the directory itself,

(2) To achieve the site autonomy objective,

(3) To avoid making the directory a source of bottleneck.

The names of the two relations that constitute a data directory are D_RELATION and D_ATTRIBUTE.

The relation D_RELATION contains the necessary descriptive information to locate all the copies and the current availability status of each relation in the DDB. It contains all the required information to provide location and replication transparencies. This information is checked by FREDI to confirm the existence and the availability of the needed relation before the actions for each query are carried out. An example of the D_RELATION looks like the following:

relid	primary	secondary	final
employee	vaxa		
projects	vaxa	vaxc	
dept	vaxb		
parts	vaxb	vaxc	
salary	vaxc	vaxd	vaxc
address	vaxd	vaxb	vaxb

•

•

•

The domains in the relation D_RELATION contain the following information:

relid
The names of the relations.

primary
The name of the site holding the primary copy of each DDB relation.

secondary
The name of the site holding the secondary copy of each relation. If a relation does not have a secondary copy, this entry is empty.

final
The name of the site holding the "final copy" of each relation. There will be an entry to this domain only if the relation has two data representations, and currently one or both of the data representations are not available. The above example displays the entries to the domain when the site "vaxd" is down.

The relation D_ATTRIBUTE contains the lower level "format" information about the domains of all of the FREDI DDB relations. In fact, the D_ATTRIBUTE relation is really the union of the local INGRES ATTRIBUTE relation at all of the DDB sites. The ATTRIBUTE relation is part of INGRES's data directory.

The existence of a copy of D_ATTRIBUTE at each site makes the low level descriptive information about any user DDB relation available locally. This significantly enhances the site-autonomy objective; during the time an input query is parsed, the FREDI master site is completely self sufficient and no inter-site data transmission is necessary. This eliminates the inconvenience of making a remote request every time detailed information about a remote relation is required. An example of the D_ATTRIBUTE looks like the following:

attrelid	attname	attfrmt	attfrml
salary	name	c	12
salary	salary	i	4
employee	name	c	12
employee	position	c	12
employee	emp_id	i	2
address	name	c	12
address	street	c	20
address	city	c	12
address	code	c	7

⋮

The domains in the D_ATTRIBUTE contain the following information:

attrelid
The name of the relation the domain belongs to.

attname
The name of the domain.

attfrmt
The type of the attribute (i.e. binary or character).

attfrml
The length of the domain (in bytes).

If any statistical information, such as the number of tuples of each relation, is desired (due to a more elaborate query processing method being used, etc.) to be maintained by the directory, this information may be added as new domains to the D_RELATION relation. Rather than updating these statistical domains immediately every time the questioned relation is altered, it is suggested that they be updated periodically instead. How often this statistical information should be updated should be proportional to how dynamic are the relations in question. This would be a usage problem, rather than a DDB design problem. In this case, the DDB system should enable the users to specify how often the statistical information should be updated, using a syntax such as the following:

update_stat sale_order every 2 hours

If the user did not specify the frequency of update, a default time value, such as every 24 hours, should be set to update the statistical information.

Fully replicating the data directory will inevitably involve more overhead during directory updates. However, if site autonomy is a very high priority, there seems to be no obvious alternative to fully replicating the data directory. Fortunately, with a close look at the content of FREDI's data directory, one will discover that the content of the data directory should be fairly stable. Even if some statistical information is incorporated as part of the data directory, if this information is updated periodically as just described, rather than updated perpetually, the data directory should still remain fairly stable.

6.5. SITE RECOVERY IN FREDI

When a crashed site **A** in a FREDI environment recovers, it must assume that the copy of the data directory stored at site **A** is obsolete; it no longer reflects the current condition of the DDB or the network. While site **A** was out of service, new relations may have been created, old relations may have been destroyed, crashed sites may have become operational again, and operational sites may have crashed, etc. Therefore, the first step to site recovery is to ship in a fresh copy of the data directory from an operational site to site **A**. Since all operational sites store an identical (current) copy of the data directory, any operational site can supply site **A** with a copy of the current data directory. The choice of the supplier site is simple: it can either be chosen by random or by some kind of ordering of sites.

However, one should be cautious of the following: At all times, within a FREDI system, at least one site must survive in order that the data directory be kept up to date. Since the data directory is fully replicated, as long as one site survives, the *last copy* of the data directory is preserved. If at one time all sites in the system have crashed, it would be impossible to tell which site is holding the last version of the data directory.

Once site **A** has acquired a current copy of the data directory, it examines the data directory and can now carry on to perform recovery on user data.

To explain how user relations are recovered, let us assume that relation **X** has representations in both site **A** and site **B**. When site **A** is detected failed, FREDI marks in its data directory (at all operational sites) that the remaining copy at site **B** is the *final* copy of relation **X**. From now on, all updates to relation **X** are directed to the final copy (at site **B**). The final copy is the most up-to-date. When site **A** recovers, it copies the value of relation **X** from the final copy (at site **B**). When this is done, the recovery manager at site **A** removes the final copy marker of relation **X** in the data directory (at all sites).

If however, when site **A** recovers, it discovers that site **B** is down, and therefore the final copy of relation **X** is not available, it must defer the recovery of relation **X** until site **B** is up again. Since the final copy is most up-to-date, it must not risk losing the changes made to relation **X** during the time when site **A** is down. In this case relation **X** will remain unavailable until site **B** recovers.

The idea of the scheme described was inspired by Bernstein and Goodman [Bern84]. It works best when there are two copies. When there are more than two copies, it is much more complex to keep track of which site (or sites) holding a copy of the relation failed last, and therefore more complex to determine which copy is the final copy. Using the basic idea of "final copy", to tolerate n-site failures, the system needs only $n+1$ copies. Consequently, FREDI is capable of tolerating single-site failures.

It should be mentioned that site recovery is necessary only because data are replicated. The purpose of site recovery is to bring replicated data to a consistent state. If the data are not replicated, no site recovery is necessary. To deal with situations where a site crashes during the middle of an update to a relation, it would be mainly the responsibility of the local data manager at that site rather than the responsibility of the DDB system. The local data manager must in this case guarantee either to run every local update to completion, or not to perform the update at all. In FREDI, this responsibility lies within INGRES. Referring to section 2.3.5, the deferred update mechanism in INGRES should probably handle this quite well. As a matter of fact, even if it wants to, there is no way for a DDB system at the front end level to guarantee the completion of a local update during a site failure. Fortunately, thanks to the simple syntax in QUEL, every QUEL statement can update at most only one relation. Furthermore, a site failure during the middle of a local update does not occur very often.

6.6. UPDATE PROPAGATION PROBLEM IN FREDI

When a relation has two stored representations, an update to the relation is always first directed to the primary copy, and then to the secondary copy. When there is no site failure, the operation is very simple.

One may wonder what might happen if during an update to a replicated relation either the primary site or the secondary site of the relation crashes before the update can be completed at both sites. Let us assume the following situation: The primary copy of the relation **X** is stored at site **A**, and the secondary copy is stored in site **B**. An update is required to be performed on relation **X**. A site failure can occur in the following four scenarios:

(1) During the update to the primary copy, site **A** fails. When this occurs, the transaction cannot proceed to update the secondary copy of **X**, because it never gets a "complete" message from site **A**. The net effect in this case would be as if the update had never started. The transaction is lost.

At first glance, one may think that it may be possible to perform a time out on site **A**. The transaction should move on to update the copy of **X** at site **B** if site **A** does not reply, so that the update will not be lost. However, this is not as simple as it may appear, for the following reasons:

First, time out on a remote update is very difficult. It is extremely difficult (if possible at all) to predict how long it will take for the remote site to complete the required update.

Second, a transaction acquires its knowledge on the survival status of all the sites at the beginning of the transaction. For it to find out that the copy of **X** at site **B** has changed its status and has became the final copy after the transaction has already started, the transaction has to perpetually monitor the survival status of all sites. This can be done, but is very expensive. Since it is rare that a site crashes in the middle of a transaction, this effort does not seem to be worthwhile.

Third, locking of relations would also be a problem. Since a lock on a relation is acquired at either the primary site or the final site (whichever one applies under the circumstances) of the relation at the beginning of the transaction, when site **A** crashes and site **B** becomes the final site of **X**, the lock on **X** is lost at that moment. In order to update relation **X** at site **B**, a new lock must be acquired at site **B** again. However, if things are allowed to proceed this way, the serializability of transactions cannot be guaranteed. Also, deadlocks between transactions can happen (FREDI prevents the occurrence of deadlocks by not allowing a transaction to begin until it has acquired the locks on all required relations).

(2) During the update to the primary copy, site **B** fails. In this case, the update to the primary copy of **X** will still run to completion. When site **B** recovers, the recovery routines copy the value of **X** from site **A** to site **B**. The update to **X** eventually applies to both the primary copy and the secondary copy of **X**.

(3) Update to the primary copy is completed, but during the update to the secondary copy, site **A** fails. In this situation, update to the secondary copy of **X** will still run to completion.

(4) Update to the primary copy is completed, but during the update to the secondary copy, site **B** fails, and the update to the secondary copy cannot run to completion. When site **B** recovers, the recovery routines copy the value of **X** from site **A** to site **B**. The update eventually applies to both copies of **X**.

Therefore, an update to a replicated relation will either eventually be applied to both the primary copy and the secondary copy of the relation, or the update will not be applied at all. The copies of a relation will not be in an inconsistent state.

6.7. CONCURRENCY CONTROL IN FREDI

FREDI uses locking, rather than timestamping, for concurrency control. This is for the following reasons:

(1) Locking appears to be more straight forward to implement.

(2) Message overhead due to locking is not an extremely important factor in a local area network environment.

(3) It is cumbersome to need to attach a clock reading to each data object (i.e an INGRES relation).

(4) To synchronize all the clock readings across all sites would require a special algorithm. Clock synchronization is needed in most timestamping schemes.

(5) A timestamping scheme may occasionally require *rollback* of a transaction. The prototype FREDI is not equipped to perform any rollback of transactions. In fact, it is extremely difficult to design a DDB system at the front-end level that could perform any rollback of transactions.

Since it is not possible to apply a physical lock to a relation, logical lock is used instead. For relations that have only a single representation, all locks are handled by the lock server at the site where the relation resides. For relations that have two stored representatives, all locks are directed to the primary site if both copies are available, and to the "final" site (either primary site or secondary site) if one of the two copies is not available. Therefore, locking in FREDI is handled in a distributed fashion. The centralized locking scheme is not used because it is prone to the the failure of the central control site as well as the possibility of a bottleneck.

The locking status of the relations at each site is maintained by a local relation called D_LOCK. Thus, there is one D_LOCK relation at each site. Each D_LOCK relation maintains only the locking status of the user relations at that site.

Except for the local lock server, no other process should have direct access to D_LOCK. D_LOCK has three domains: **rel_id** is the name of the relation being locked, **p_label** registers who (which user process group) owns the lock (p_label stands for "process label"), and **type** registers what type of lock (read or write) is granted. An example of the D_LOCK relation looks like the following:

rel_id	p_label	type
employee	3	r
salary	3	w
projects	7	r
parts	7	r
managers	3	r
depts	2	w
customers	3	r
projects	4	r

•
•
•

Only the names of those relations that are currently locked appear in the relation D_LOCK.

When a request for locking a relation arrives, the lock server first checks the D_LOCK relation to see if the request can be honored. If it can, the lock server process adjusts the D_LOCK relation to reflect the change, and returns a message to the requestor to indicate that the operation has been completed. Otherwise, a message indicating that the request cannot be completed is returned to the requestor. The lock server may be considered to perform "test-and-set" operations. It determines if a lock should be granted by following these rules:

- If a user process group requests a *read* (shared) lock on a relation, and a *write* (exclusive) lock on that relation has not been granted to another user process group, grant the lock.
- If a user process group requests a *write* lock on a relation, and neither a *read* lock nor a *write* lock has been granted to another user process

group, grant the lock.

- Deny the request otherwise.

According to the query processing strategy of the prototype FREDI, there is no need to enable a read lock to be upgraded to a write lock, or a write lock to be downgraded to a read lock. The correct locks are requested before the actions for the current query are carried out. No further lock is required once the actions for the query have been started.

Deadlocks in the prototype FREDI are avoided. Since it is very difficult for a front-end level DDB system to perform rollback of transactions, other methods to deal with deadlock do not seem to be appropriate in FREDI. The basic idea is that a transaction cannot proceed until it has locked all the required relations. Once a lock on a relation cannot be granted, the transaction must relinquish all the locks it has already obtained, and try to lock all of the required relations again after a random period of time. The locking of relations for a transaction therefore follows these steps:

(1) Form a queue that contains all the names of the relations that are needed for this transaction.

(2) Pop the next name off the queue. If the queue is empty, all the locks needed have been acquired. Otherwise, the name obtained is the name of "the relation in question".

(3) Request an appropriate lock for the relation in question.

(4) If the lock for the relation in question can be obtained, go to step 2.

(5) If the lock for the relation in question cannot be obtained, release all the locks already acquired. Re-form the queue so that it contains all the

names of the relations that are needed for this transaction. Wait a random period of time (such as one, two, or three seconds). Go to step 2.

6.8. TRANSACTION SCOPE AND COMMIT PROTOCOL

In FREDI, commit protocol such as two-phase commit is not needed.

Recall that a single QUEL statement is defined as a single transaction. On close observation, one will discover that among all QUEL statements there is no single statement that can update more than a single relation at a time. Commit protocols are needed when different data in more than one site are needed to be updated within a single transaction. Since there is no need to update more than one relation within a single transaction in FREDI, there is no need for a commit protocol.

CHAPTER 7

CONCLUSION

The development of FREDI demonstrates that it is possible to construct a DDB system using a front-end approach. FREDI is easy to use, provides much site autonomy, gives good performance to single site operations and reasonable performance to multi-site operations, and tolerates single site failures.

In the development of FREDI, we have also explored many design issues about DDB system in general.

The subject of DDB systems is still very new, and many issues do not have a "correct" solution. During the development of FREDI, particularly in the early stages, looking for the appropriate solution was occasionally like "searching in the dark". Fortunately, many problems became much clearer and better understood as FREDI started to take shape. Even so, FREDI had to be re-written from scratch several times during the early course of this research.

There are a few issues that we either have not touched at all, or have not explored very deeply. Possible avenues of further research include:

(1) Security and authorization.

FREDI may be extended as a vehicle for research in issues regarding data security, data authorization, user and remote site authentication in DDB environments.

(2) View management.

FREDI does not support **view**. View management in DDB systems

[Bert83] is not as simple as it first appears. Future research could add the support of view to FREDI.

(3) Data partitions.

FREDI supports neither horizontal partition nor vertical partition for the reasons we have explained. While the support of horizontal partition and/or vertical partition would require very serious effort, both kinds of partitions are nevertheless interesting forms of data distribution, and merit further study.

(4) Query optimization.

The prototype FREDI does not use a fancy algorithm for query processing nor does it handle queries in a truly distributed fashion. This is acceptable only in a local area network environment where data communication cost and data communication delay are not significant factors. Nevertheless, future research in query optimization may employ FREDI as a testing ground.

(5) Parallelism.

The prototype FREDI does not utilize parallelism very much. Although it is much harder to control parallel distributed operations than sequential distributed operation, parallelism is undeniably an obvious way to improve performance of a DDB system.

In this research we have shown that a DDB system can be a practical vehicle for information management. It is therefore obvious that DDB systems have a bright future.

References and Bibliography

[Adib78]
Adiba, M., et al. (September 1978) "Issues In Distributed Data Base Management Systems: A Technical Overview" in *Proc. 4th International Conference on Very Large Data Base,* pp 89 - 110.

[Aper83]
Apers, Peter M. G., Alan R. Hevner, and S. Bing Yao (January 1983) "Optimization Algorithms for Distributed Queries" in *IEEE Transactions on Software Engeering, SE-9* (1) pp 57 - 68, IEEE.

[Bada79]
Badal, D. Z. (1979) "Concurrency Control and Semantic Integrity Enforcement In Distributed Databases" in *Distributed Databases, Vol. 2: Invited Papers,* pp 15 - 27. Infotech International Limited, Maidenhead, Berkshire, England, ISBN 8553-9580-X.

[Bern84]
Bernstein, Philip A. and Nathan Goodman (December 1984) "An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases" in *ACM Transactions on Database Systems, 9* (4) pp 596 - 615, ACM.

[Bert83]
Bertino, E., L. M. Haas, and B. G. Lindsay (April 8, 1983) View Management in Distributed Data Base System, IBM Research Laboratory, San Jose, California, IBM Reserch Report RJ 3851 (43918).

[Boot79]
Booth, G. M. (1979) "Distributed Databases on Unlike Computer Systems" in *Distributed Databases, Vol. 2: Invited Papers,* pp 30 - 46. Infotech International Limited, Maidenhead, Berkshire, England, ISBN 8553-9580-X.

[Brac80]
Bracchi, G., C. Baldissera, and S. Ceri (1980) "Distributed Query Processing" in *Distributed Data Bases,* edited by I. W. Draffan and F. Poole, pp 83 - 101. Cambridge University Press, Brattleborl, Vermont, ISBN 0-521-23091-8.

[Cham79]
Champine, G. A. (1979) "Distributed Data Systems Overview With Case Studies" in *Distributed Databases, Vol. 2: Invited Papers,* pp 47 - 71. Infotech International Limited, Maidenhead, Berkshire, England, ISBN 8553-9580-X.

[Chu79]
Chu, W. W. (1979) "Design Considerations of File Directory Systems For Distributed Databases" in *Distributed Databases, Vol. 2: Invited Papers,* pp 73 - 85. Infotech International Limited, Maidenhead, Berkshire, England, ISBN 8553-9580-X.

[Dani82]
Daniels, Dean, et al. (June 4, 1982) An Introduction to Distributed Query Compilation in R^*, IBM Research Laboratory, San Jose, California, IBM Reserch Report RJ 3497 (41354).

[Date83]
Date, C. J. (1983) *An Introduction to Database Systems, Vol. 2.* Addison-Wesley, Don Mills, Ontario.

[Date86]
Date, C. J. (1986) *An Introduction to Database Systems, Vol.1 (Fourth Edition).* Addison-Wesley, Don Mills, Ontario.

[Draf80]
Draffan, I. W. and F. Poole (1980) "The Classification of Distributed Data Base Management Systems" in *Distributed Data Bases,* edited by I. W. Draffan and F. Poole , pp 57 - 81. Cambridge University Press, Brattleborl, Vermont, ISBN 0-521-23091-8.

[Epst77]
Epstein, Robert (December 1977) *A Tutorial on INGRES.* Electronics Research Laboratory, University of California, Berkeley.

[Epst78]
Epstein, Robert, Michael Stonebraker, and Eugene Wong (1978) "Distributed Query Processing in a Relational Database System" in *The INGRES Paper: Anatomy of a Relational Database System,* edited by Michael Stonebraker. Addison-Wesley, Don Mills, Ontario, Also appears on ACM-SIGMOD Conference Proceedings, ACM-SIGMOD International Conference on Management of Data, Austin, Texas, May 1978. ISBN 0-201-07185-1.

[Giff85]
Gifford, David and Alfred Spector (August 1985) "The Cirrus Banking Network" in *Communication ACM, 28* (8) pp 798-807, ACM.

[Gilm82]
Gilman, Micheal and David Burrage (1982) *RAQL: Relational Algebraic Query Language User Manual.* McGill University, Montreal.

[Haas82]
Haas, Laura M., et al. (September 21, 1982) R^*: A Research Project on Distributed Relational DBMS, IBM Research Laboratory, San Jose, California, IBM Reserch Report RJ 3653 (42509).

[Hevn79]
Hevner, A. R. and S. B. Yao (May 1979) "Query Processing in Distributed Database System" in *IEEE Transactions on Software Engineering, SE-5* (3), IEEE, ISSN 0098-5589.

[Hous79]
Housley, Trevor (1979) *Data Communications and Teleprocessing Systems.* Prentice Hall, Englewood Cliffs, N.J., ISBN 0-13-197368-1.

[Jark84]
Jarke, Mattias and Jurgen Koch (June 1984) "Query Optimization in Database Systems" in *ACM Computing Surveys, 16* (2) pp 111 - 152, ACM.

[John78]
Johnson, Stephen C. (January, 1978) "Yacc: Yet Another Compiler-Compiler" in *UNIX Programmer's Manual.* Bell Telephone Laboratories, Murray Hill, New Jersey.

[Katz79]
Katzan, Harry Jr. (1979) *Distributed Information Systems.* Petrocelli Books, Inc., New York, ISBN 0-89433-104-3.

[Kern78]
Kernighan, Brian W. and D. M. Ritchie (1978) *The C programming Language.* Prentice Hall, Englewood Cliffs, New Jersey.

[Kern84]
Kernighan, Brian W. and Rob Pike (1984) *The UNIX Programming Enivronment.* Prentice Hall, Englewood Cliffs, New Jersey, ISBN 0-13-937681-X.

[Lann80]
Lann, G. Le (1980) "Consistency, Synchronization and Concurrency Control" in *Distributed Data Bases,* edited by I. W. Draffan and F. Poole, pp 195-222. Cambridge University Press, Brattleborl, Vermont, ISBN 0-521-23091-8.

[Leff83]
Leffler, Samuel J., Robert S. Fabry, and William N. Joy (July, 1983) *A 4.2bsd Interprocess Communication Primer.* University of California, Berkeley, California.

[Lesk79]
Lesk, M. E. (January, 1979) "Lex - A Lexical Analyzer Generator" in *UNIX Programmer's Manual.* Bell Telephone Laboratories, Murray Hill, New Jersey.

[Lind80a]
Lindsay, Bruce G. (1980) "Single and Multi-site Recovery Facilities" in *Distributed Data Bases,* pp 247 - 284. Cambridge University Press, Brattleborl, Vermont, ISBN 0-521-23091-8.

[Lind80b]
Lindsay, Bruce and Patricia G. Selinger (September 15, 1980) Site Autonomy Issues in R*: A Distributed Database Management System, IBM Research Laboratory, San Jose, California, IBM Reserch Report RJ 2927 (36822).

[Lind80c]
Lindsay, Bruce (August 29, 1980) Object Naming and Catalog Management for a Distributed Database Manager, IBM Research Laboratory, San Jose, California, IBM Reserch Report RJ 2914 (36689).

[Lind85]
Lindsay, Bruce (September 25, 1985) A Retrospective of R*: A Distributed Database Management System, IBM Research Laboratory, San Jose, California, IBM Reserch Report RJ 4859 (51208).

[Mart81]
Martin, James T. (1981) *Design and Strategy For Distributed Data Processing.* Prentice Hall, Englewood Cliffs, New Jersey, ISBN 0-13-201657-5.

[Mcgl78]
Mcglynn, Daniel R. (1978) "Computer Networks" in *Distributed Processing and Data Communications.* John Wiley and Sons, Toronto, ISBN 0-471-01886-4.

[Roth77]
Rothnie, J. B. Jr. and N.Goodman (October 1977) "A Survey of Reasearch and Development in Distributed Database Management" in *Proc. 3rd International Conference on Very Large Data Bases.*

[Rowe86]
Rowe, Lawrence A. and Michael Stonebraker (1986) "The Commercial INGRES Epilogue" in *The INGRES Paper: Anatomy of a Relational Database System,* edited by Michael Stonebraker. Addison-Wesley, Don Mills, Ontario, ISBN 0-201-07185-1.

[Spac80]
Spaccapietra, Stefano (1980) "Heterogeneous Data Base Distribution" in *Distributed Data Bases,* edited by I. W. Draffan and F. Poole, pp 155 - 193. Cambridge University Press, Brattleborl, Vermont, ISBN 0-521-23091-8.

[Ston76]
Stonebraker, Michael, et al. (September 1976) "The Design and Implementation of INGRES" in *ACM Transactions on Database Systems, 1* (3) pp 189 - 222, ACM.

[Ston77]
Stonebraker, Michael and Erich Neuhola (May 1977) "A Distributed Data Base Version of INGRES" in *Proceding 2nd Berkeley Conference on Distributed Data Management and Computer Networks,* pp 19 - 36, Lawrence Berkeley Laboratory.

[Ston79]
Stonebraker, Michael (May 1979) "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES" in *IEEE Transactions on Software Engineering, SE-5* (3), IEEE, ISSN 0098-5589.

[Ston80a]
Stonebraker, Michael (1980) "Retrospection on a Database System" in *The INGRES Paper: Anatomy of a Relational Database System,* edited by Michael Stonebraker. Addison-Wesley, Don Mills, Ontario, Also appears on ACM transactions on Database Systems, vol.5, no.2 June 1980. ISBN 0-201-07185-1.

[Ston80b]
Stonebraker, Michael (1980) "Homogeneous Distributed Data Base Systems" in *Distributed Data Bases,* edited by I. W. Draffan and F. Poole. Cambridge University Press, Brattleborl, Vermont, ISBN 0-521-23091-8.

[Ston86]
Stonebraker, Michael (1986) "The Design and Implementation of Distributed INGRES" in *The INGRES Paper: Anatomy of a Relational Database System,* edited by Michael Stonebraker. Addison-Wesley, Don Mills, Ontario, ISBN 0-201-07185-1.

[Taki80]
Takizawa, Makoto and Eiji Hamanaka (1980) "Query Translation in Distributed Databases" in *Information Processing 80,* edited by S. H. Lavington, pp 451 - 456. North-Holland, ISBN 0-444-86034-7.

[Tayl80]
Taylor, Frank E. (1980) "Distributed Systems In Perspective" in *Distributed Data Bases,* edited by I. W. Draffan and F. Poole, pp 1 - 31. Cambridge University Press, Brattleborl, Vermont, ISBN 0-521-23091-8.

[Tech80]
Techo, Robert (1980) *Data Communications: An Introduction to Concepts and Design.* Plenum Press, New York, ISBN 0-306-40398-6.

[UNIX84]
(August, 1984) *UNIX programmer's manual.* Deparment of Computer Science, University of Calgary, Calgary.

[Won82]
Won, Kim (September 1982) "On Optimizing an SQL-like Nested Query" in *ACM Transactions on Database Systems, 7* (3), ACM.

[Wong76]
Wong, Eugene and Karel Youssefi (September 1976) "Decomposition — A Strategy for Query Processing" in *ACM Transactions on Database Systems, 1* (3) pp 223 - 241, ACM.

[Wong83]
Wong, Eugene (1983) "Dynamic Rematerialization: Processing Distributed Queries Using Redundant Data" in *The INGRES Paper: Anatomy of a Relational Database System,* edited by Michael Stonebraker. Addison-Wesley, Don Mills, Ontario, Also appears on IEEE Transactions on Software Engineering, vol. SE-9, no.3, pp.228-232, May 1983. ISBN 0-201-07185-1.

[Wood79]
Woodfill, John, et al. (March 1979) *INGRES Version 7 Reference Manual.* Electronics Research Laboratory, University of California, Berkeley.

[Yost85]
Yost, Robert A. and Laura M. Haas (April 29, 1985) R^*: A Distributed Data Sharing System, IBM Research Laboratory, San Jose, California, IBM Reserch Report RJ 4676 (49844).

[Yous79]
Youssefi, Karel and Eugene Wong (1979) "Query Processing in a Relational Database Management System" in *The INGRES Paper: Anatomy of a Relational Database System,* edited by Michael Stonebraker. Addison-Wesley, Don Mills, Ontario, Also appears on Proceedings of the Fifth Very Large Data Base Conference, Rio de Janeiro, 1979. ISBN 0-201-07185-1.

[Yu84]
Yu, C. T. and C. C. Chang (December 1984) "Distributed Query Processing" in *ACM Computing Surveys, 16* (4) pp 399 - 433, ACM.

[Yu85]
Yu, Clement T., et. al. (August 1985) "Query Processing in a Fragmented Relational Distributed System: Mermaid" in *IEEE Transactions on Software Engeering, SE-11* (8) pp 795 - 810, IEEE.