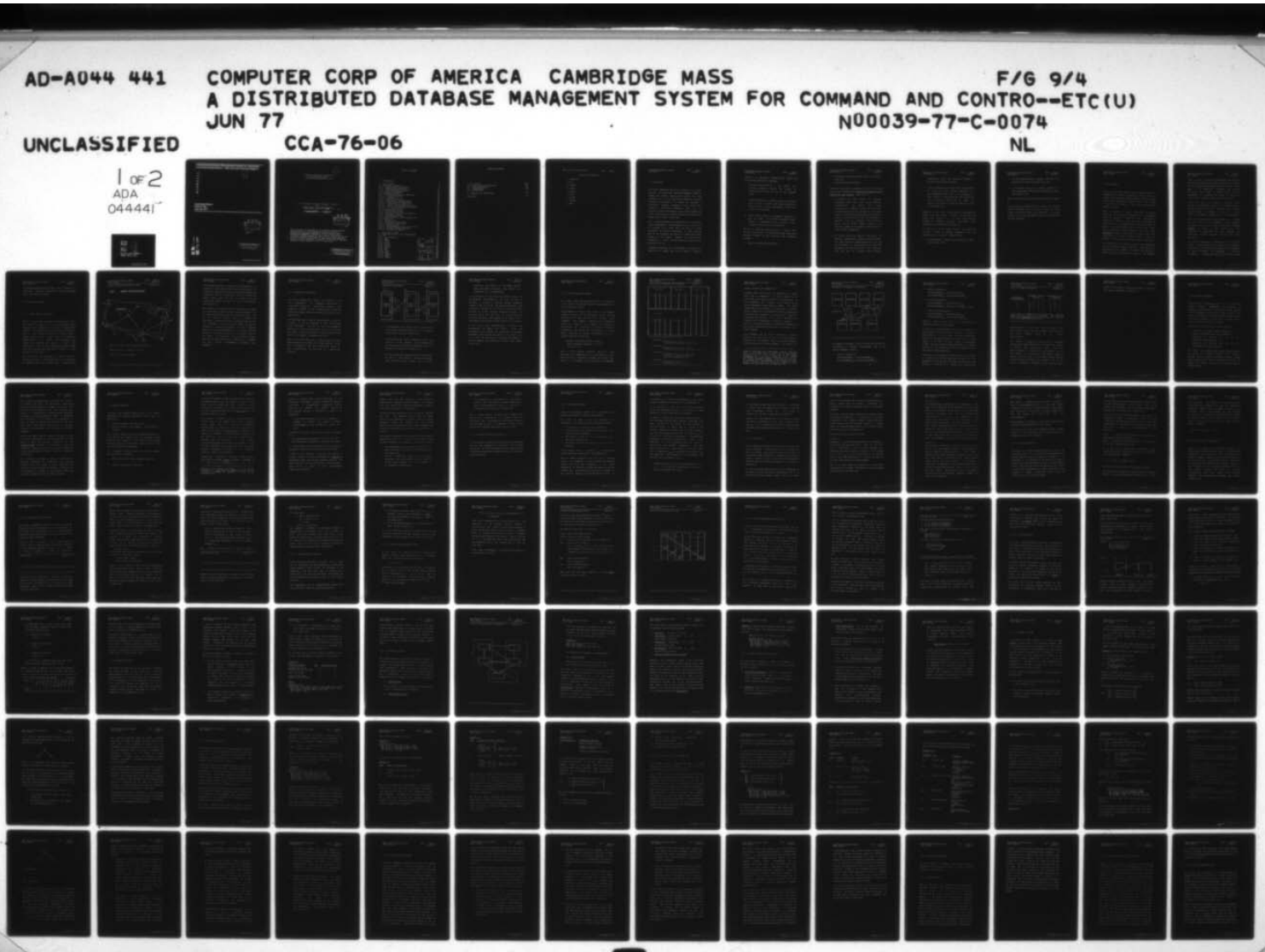
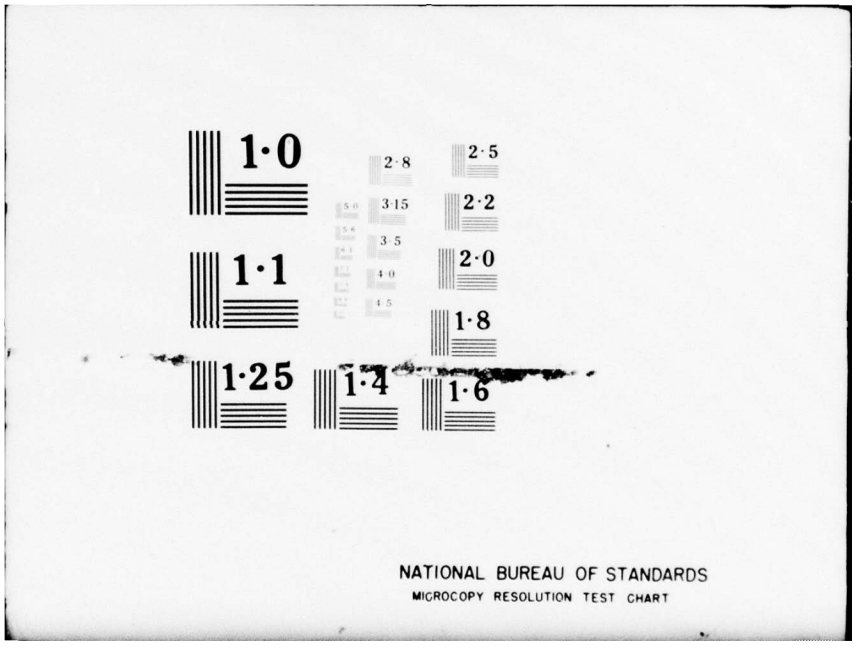
AD-A044 441 COMPUTER CORP OF AMERICA CAMBRIDGE MASS F/G 9/4
A DISTRIBUTED DATABASE MANAGEMENT SYSTEM FOR COMMAND AND CONTRO--ETC(U)
JUN 77 N00039-77-C-0074
UNCLASSIFIED CCA-76-06 NL

1 OF 2
ADA
044441




1·0
2·8
2·5
3·15
2·2
3·5
2·0
4·0
4·5
1·8
1·25
1·4
1·6
NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

A Distributed Database Management System for Command and Control Applications: Semi-Annual Technical Report I

AD A044441

(12)

Technical Report
CCA-77-06
July 30, 1977

DDC
RECEIVED
SEP 22 1977
A

AD No.
DDC FILE COPY

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

Computer Corporation of America

Computer Corporation of America
575 Technology Square
Cambridge, Massachusetts 02139

A Distributed Database Management System for Command and Control Applications

SEMI-ANNUAL TECHNICAL REPORT I

January 1, 1977 to June 30, 1977

DDC
RECEIVED
SEP 22 1977
A

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by the Naval Electronic Systems Command under Contract No. N00039-77-C-0074. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

Table of Contents

1. Introduction	2
2. SDD-1 Design	7
2.1 Overview of Design	9
2.1.1 Global System Architecture	9
2.1.2 Datamodule Architecture	12
2.1.3 Data Distribution Concepts	15
2.1.4 Directory Management	22
2.2 Redundant Updating	24
2.2.1 Protocols and Protocol Selection	28
2.2.1.1 Primitive Actions	29
2.2.1.2 Protocols	31
2.2.1.3 Protocol Selection	34
2.2.2 Safe Protocol Configurations	36
2.2.2.1 Global Logs and Consistency	37
2.2.2.2 Constructing a Global Log	40
2.2.2.3 Global Log Transformation Rules	41
2.2.2.4 Serially Reproducible Global Logs	45
2.2.2.5 L-U Graph Analysis	48
2.3 Dispersed Data Access	52
2.3.1 A Difference of View	55
2.3.2 The Basic Tactics of Distribution	59
2.3.3 A Strategy of Moves	62
2.3.4 Determination of M(Q)	67
2.3.5 Node Splitting in the Strategy Tree	72
2.4 Reliability	79
2.4.1 Introduction	79
2.4.2 Structure of the Problem	83
2.4.3 Reliability mechanisms	89
2.4.3.1 Reformulation	89
2.4.3.2 Synchronization with a Failed GDM	91
2.4.3.3 Reliable Message Delivery	92
2.4.3.4 Broadcast of Updates	93
3. Activities at NOSC	95
3.1 Installation of Initial Datacomputer	95
3.2 TAP	96
3.2.1 Initialization	97
3.2.2 ADD	97
3.2.3 CHANGE	98
3.2.4 COUNT	99
3.2.5 CREATE	99
3.2.6 DESTROY	100
3.2.7 DISPLAY	100
3.2.8 HELP	101
3.2.9 JOIN	102
3.2.10 LIST	103
3.2.11 MOVE	104
3.2.12 PROJECT	105
3.2.13 QUIT	105
3.2.14 REMOVE	106

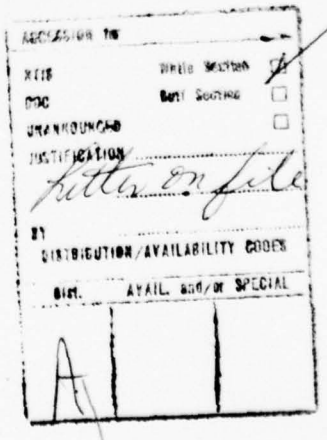


Table of Contents

3.2.15 SELECT	106
3.3 Initial Alerting Capability	107
3.4 TOPS20 Datacomputer	109
3.5 Consultation	110
3.6 Documentation	111
4. Meetings and Publications	112
References	114

Project Staff Members:

P. BERNSTEIN

S. FOX

N. GOODMAN

M. HAMMER

T. LANDERS

C. REEVE

J. ROTHNIE

D. SHIPMAN

G. WONG

1. Introduction

This report summarizes the first six months of a project entitled, "A Distributed Database Management System for Command and Control Applications" which has been undertaken by CCA and sponsored by ARPA-IPTO. The key objective of this effort is to design and implement a distributed database management system called SDD-1 (System for Distributed Databases). SDD-1 will be installed in phases and tested in the Advanced Command and Control Architectural Testbed (ACCAT) at the Naval Ocean Systems Center (NOSC) in San Diego.

SDD-1 is being designed as a cooperating set of processors called "datamodules." Each datamodule stores a portion of the database and a given data item may be stored redundantly at several datamodules. A user accessing SDD-1 at any datamodule can retrieve or update data stored anywhere in the network (subject to authorization limitations). In fact, the user need not be aware of the distribution at all.

A database system with this distributed architecture is important for command and control beause it supports

access to an integrated database while offering the following key operational advantages:

1. Reliability/survivability - the system will continue operation despite the failure or inaccessibility of one or more of its database sites.

2. "Tunable" efficiency - users are able to distribute data in such a manner that portions which are heavily used in a given geographical region are stored near that region.

3. Modular upward scaling - as databases increase in size and usage, it is possible to augment system capacity to accomodate these increases by the incremental addition of new datamodules.

During the reporting period, January 1 - June 30, 1977, the SDD-1 project was in a concentrated design phase which focussed primarily on the following three technical problems:

- updating redundantly stored data;

- processing queries which access data at multiple sites; and
- achieving reliable operation.

In each of these areas a substantial portion of the design task has been completed and important new computer science results have been achieved. Specifically:

1. A mechanism has been divised for updating redundantly stored data which avoids excessive synchronization and permits most update transactions to execute as quickly as retrievals. This mechanism involves an analysis of anticipated classes of transactions which identifies potential conflict situations at the time the database is designed. This can frequently avoid the need to detect conflict during system operation and therefore speeds up most update transactions by at least an order of magnitude.

2. To process multi-site queries efficiently an heuristic optimization scheme has been developed. This algorithm treats communication delays as the key bottleneck in system operation and takes the minimization of this cost as its first objective. With the set of solutions which minimize

communication costs, the algorithm attempts to minimize local processing delays.

3. A set of principles and mechanisms has been devised to allow for the reliable operation of a distributed data base system in the presence of site failures. These mechanisms are "tunable" so that a particular application of SDD-1 can trade-off the need for reliability against its cost in proceesing power.

Summaries of the SDD-1 design results are presented in section 2 of this report. Section 2.1 overviews the design while Sections 2.2, 2.3, and 2.4 deal with the problems of redundant updating, distributed query processing, and reliability, respectively.

In addition to SDD-1 design efforts, the project has provided a series of database management tools for immediate use in the ACCAT. These include:

1. the Datacomputer, installed and operational at NOSC since January, 1977;

2. the TAP (Terminal Access Program), installed and operational at NOSC since March 1977; and

3. a preliminary version of an alerting feature for the Datacomputer, installed and operational at NOSC since March, 1977.

These service activities at NOSC are described in Section 3.

CCA has made a concerted effort to discuss the research results developed under the SDD-1 project in public forums, particularly within the Department of Defense. Section 4 lists the meetings and publications where SDD-1 has been presented.

2. SDD-1 Design

SDD-1 is a distributed database system being designed and implemented by Computer Corporation of America (CCA) in a project sponsored by the Advanced Research Projects Agency (ARPA) of the Department of Defense. Work on the system is currently in progress. This section describes the system's design.

SDD-1 is designed to support databases distributed world-wide over hundreds of database sites. The database sites are assumed to communicate over heterogeneous communication channels which may vary in bandwidth and delay, and it is not assumed that all sites are able to maintain continuous communication with each other. Also, SDD-1 is designed to support databases which are stored redundantly, meaning that some or all logical data items can be stored at multiple database sites in order to enhance the reliability and responsiveness of the system.

It is important to note the key role which data redundancy plays in achieving these objectives. Reliability and survivability are enhanced since SDD-1 can continue to access critical data items even if some of the database

sites at which the data are stored become inaccessible. Efficiency is enhanced since data items which are accessed by widely separated user communities can be stored nearby each of the communities. Redundancy also enhances scalability since growth in database usage can be accommodated by increasing the number of data copies rather than by increasing the speed of memory units.

On the other hand, data redundancy introduces severe problems in performing updates in a consistent manner. The approach taken by SDD-1 in handling these redundant update problems is addressed in Section 2.2 and in [ROTHNIE and GOODMAN], [ROTHNIE et al], and [BERNSTEIN].

Other distributed database system designs (e.g., [ALSBERG and DAY], [ELLIS], and [THOMAS-b]) have permitted redundantly stored data, but have required that the entire database be stored at every database site. This restriction is unrealistic for large databases and effectively precludes "tunable" efficiency and upward scalability.

This section presents key characteristics of the SDD-1 design: the architecture of the system and the framework used by SDD-1 for defining the distribution and redundancy of logical databases; the management of directory information; the approach taken in SDD-1 to the problem

of updating redundantly stored data; SDD-1's approach to efficiently processing retrievals involving dispersed data; and reliability issues.

2.1 Overview of Design

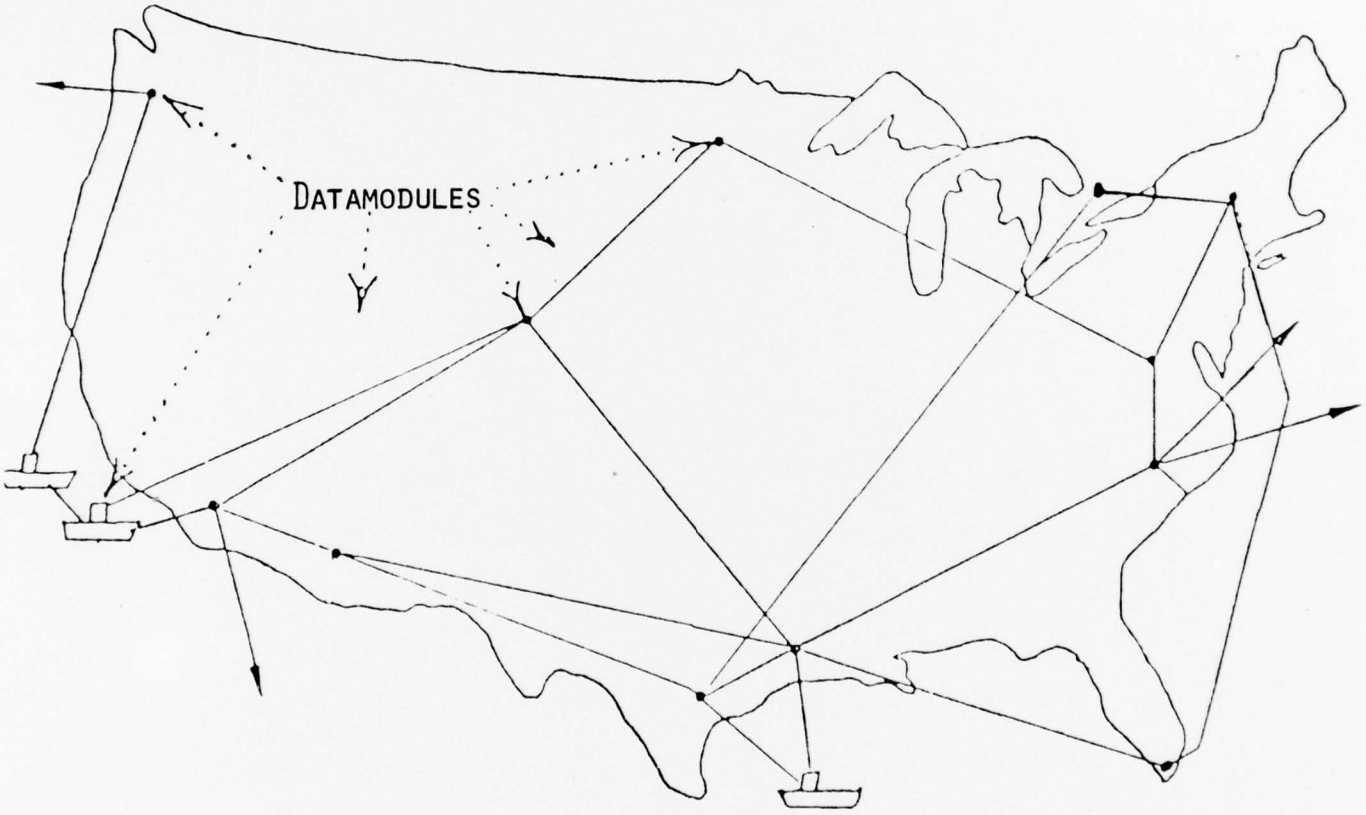
2.1.1 Global System Architecture

Figure 2.1 illustrates the general architecture of SDD-1. The system is composed of a set of components called datamodules which communicate over assorted communication channels. The datamodules are all functionally identical; that is, they respond identically to external stimuli. However, there may be a great variety of datamodule implementations. For example, some datamodules may support mass memories and operate as major data repositories in the net while others may be small systems with little storage, operating as data caches for quick response to nearby users.

The distribution and redundancy of data in SDD-1 is invisible to users. A user connecting to any datamodule is presented with the illusion that a complete and

General System Architecture of SDD-1 Figure 2.1

SDD-1 GENERAL SYSTEM DESCRIPTION



- SDD-1 COMPRISED OF DISTRIBUTED DATAMODULES.
- DISTRIBUTION IS INVISIBLE TO USERS.
- DATA IS STORED REDUNDANTLY.

non-redundant database is resident at that datamodule. In response to users' queries SDD-1 takes responsibility for locating the desired data among the distributed network of datamodules and for updating all copies of changed data items; ideally users are able to interact with the distributed system as easily as with a conventional, centralized one.

The objective of relieving users of the need to be aware of distribution issues is similar to goals pursued in the distributed operating systems being developed today, e.g. by the National Software Works (cf [SCHANTZ and MILLSTEIN]), the National Bureau of Standards NAM project (cf [ROSENTHAL]), and the Arpanet RSEXEC project (cf [THOMAS-a]). These projects take a collective computing resource distributed geographically on a computer network and make it available to users in an integrated, easy to use manner. This approach in the distributed database area has also been discussed by [ALSBERG et al], [ALSBERG and DAY], [ELLIS], [STONEBRAKER and NEUHOLD], and [THOMAS-b].

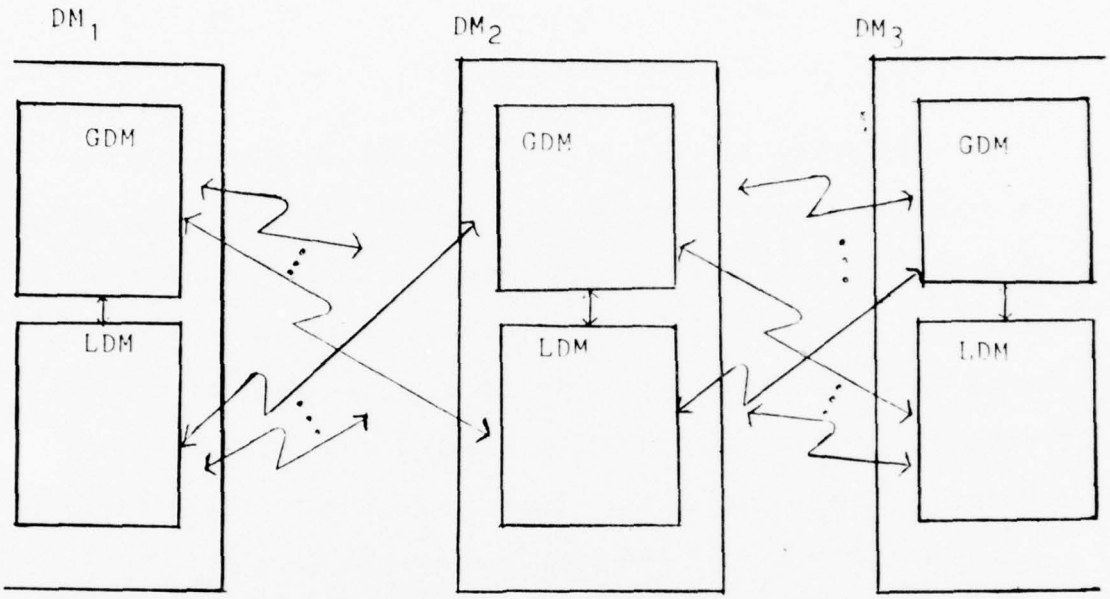
2.1.2 Datamodule Architecture

This section examines the internal architecture of the SDD-1 datamodules. The datamodule architecture is illustrated in Figure 2.2. As the figure indicates, each datamodule consists of two internal modules called the global data manager (GDM) and the local data manager (LDM) respectively.

The LDM manages data located at the datamodule of which it is a part. It has no awareness of data distribution issues whatsoever. The LDM can apply data management operators to its locally stored data, and it can retrieve and modify locally stored data. The LDM performs these functions in response to requests from GDM's.

GDM's, on the other hand, have no local storage at all and depend entirely on the LDM's for storage resources. Their role is to handle all the data distribution issues in SDD-1. Specifically, the functions of a GDM are the following:

Datamodule Architecture Figure 2.2



- The GDM parses a user request into an internal form which exposes what operations are to be performed and what data is to be operated upon.

- The GDM determines which datamodules house data involved in the requests. This determination involves access to directory information which is managed as we describe in Section 2.1.4.

- The GDM decides what operations should be performed by LDM's in the network. The decision process takes two factors into account: one is access

efficiency (see Section 2.3 and [WONG]) and the other is database consistency (see Section 2.2, [ROTHNIE et al] and [BERNSTEIN]).

A key motivation for selecting this GDM/LDM structure for the datamodule architecture is the long run goal of employing a variety of existing database management systems in the role of LDM. The LDM operations have been consciously confined to the level of function currently provided by typical database management systems. By equipping GDM's with the rules for constructing their primitive LDM calls in the language of the target LDM, a heterogeneous distributed database system can be achieved.

In the initial SDD-1 implementation all LDM's will be Datacomputers (cf [MARILL and STERN]). Since the Datacomputer was designed and implemented as a database management system without any regard for its future role as an LDM, the feasibility of using the Datacomputer in this way suggests the plausibility of using other DBMS's in this role.

2.1.3 Data Distribution Concepts

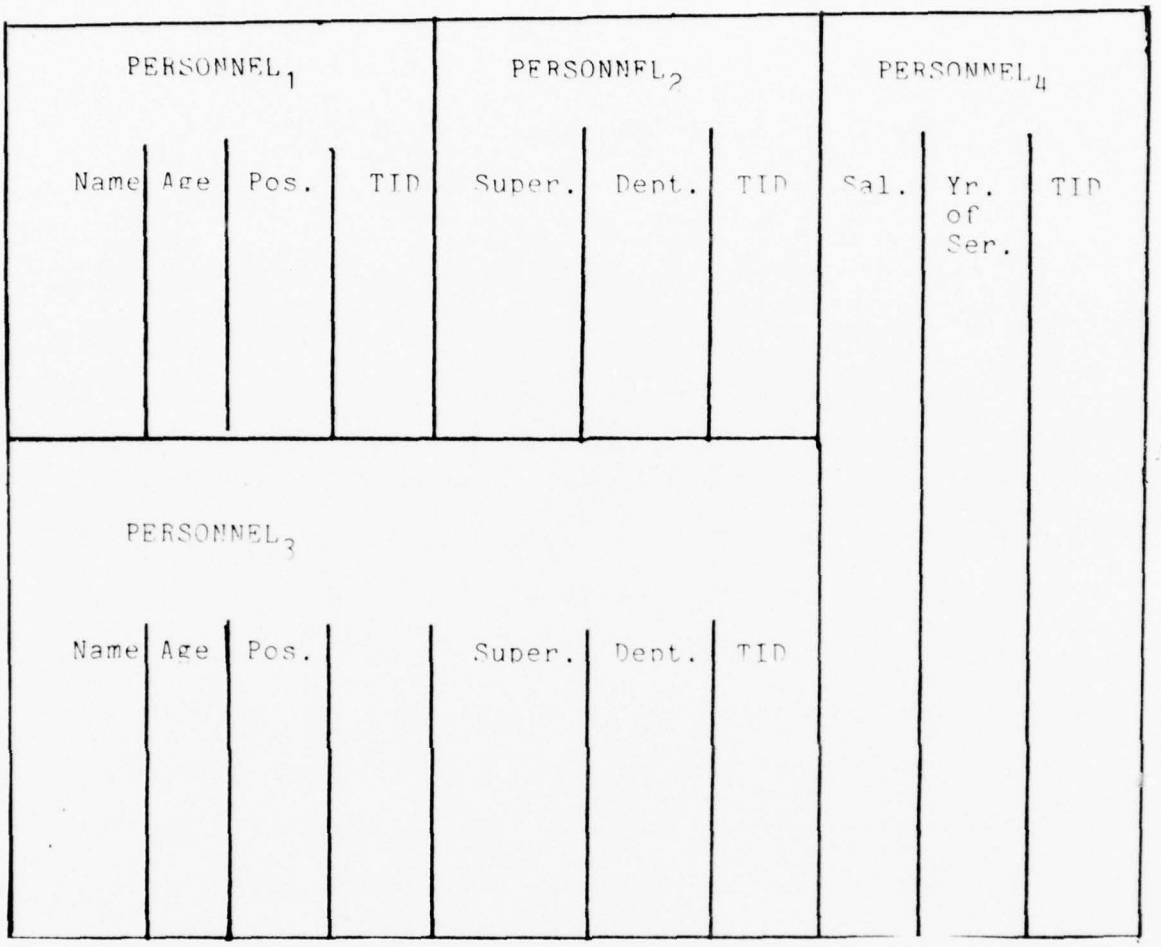
The logical data model supported by SDD-1 is relational. In this section we consider the stored representation of a set of relations in SDD-1.

The assignment of logical data items to the physical storage resources of the datamodules begins with the partitioning of each relation into sub-sets called fragments. Each fragment is defined to be a rectangular sub-set of a relation; i.e. fragments are defined as restrictions and projections of database relations. Furthermore, restrictions are constrained to involve simple predicates as defined by [ESWARAN et al], i.e., boolean conditions of the form:

> Attribute <relational operator> constant
>
> where <relational operator>:= "=","<",
> ">","<", or ">".

Also, to avoid updating anomalies similar to those described by [CHAMBERLIN et alb] with respect to views, each projection is required to include the primary key. Figure 2.3 illustrates the partitioning of a PERSONNEL

Partition of a Relation into Fragments Figure 2.3



Each fragment is defined as follows:

- $PERSONNEL_1$:=
  PERSONNEL where Salary > $30,000,
  projected on Name, Age, Position, TID

- $PERSONNEL_2$:=
  PERSONNEL where Salary > $30,000,
  projected on Supervisor, Department, TID

- $PERSONNEL_3$:=
  PERSONNEL where Salary <= $30,000,
  projected on Name, Age, Position,
  Supervisor, Department, TID

- $PERSONNEL_4$:=
  PERSONNEL projected on Salary,
  Years-of-service, TID

logical relation into fragments.(1)

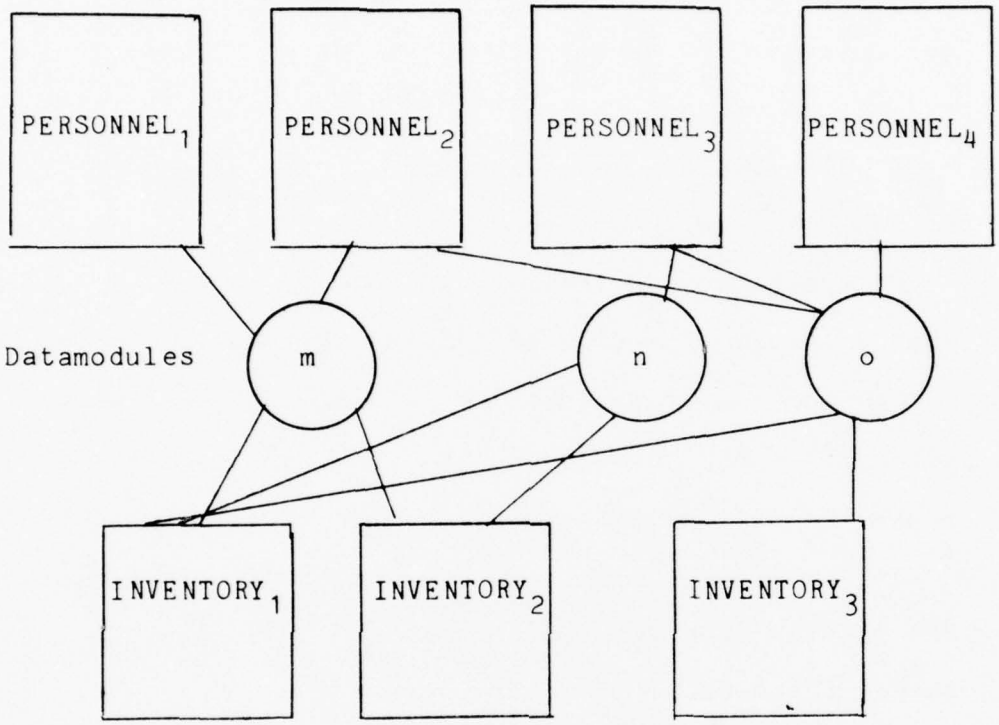
Fragments then, are the units of assignment of logical data items to datamodules. A given fragment is, by definition, either entirely present or entirely absent at each datamodule, DM_i. However, each fragment may be stored redundantly at more than one module. The stored representation of a fragment in a given module is termed a stored fragment and is designated Stored-$F_{i,m}$ meaning the representation of fragment F_i in datamodule m. Figure 2.4 illustrates the partition of a logical database into fragments and the assignment of fragments to modules. Each arc from the rectangles (representing fragments) to the circles (representing datamodules) corresponds to a stored fragment.

Since fragments may be stored redundantly, in general there will be more than one way of reconstructing a complete and non-redundant copy of the logical database from the collection of stored fragments. For instance, in

---

(1) A way of achieving this requirement without impacting users is to include in every relation an attribute called TID (for tuple ID) which can be used by the system internally but is not visible to users. INGRES [STONEBRAKER et al] and System R [ASTRAHAN et al] use the TID concept for other purposes. Like those uses, though, we assume that TID is guaranteed to be unique for each tuple. In this way, the primary key required in each fragment-defining projection can simply be TID.

Assignment of Fragments to Datamodules Figure 2.4



the example of Figures 2.3 and 2.4, there are four ways of reconstructing a complete, non-redundant copy of the logical PERSONNEL relation:

1. One which consists of

   Stored-PERSONNEL$_{1,m}$, Stored-PERSONNEL$_{2,m}$,

   Stored-PERSONNEL$_{3,n}$, and Stored-PERSONNEL$_{4,o}$;

2. Another consisting of

   $\text{Stored-PERSONNEL}_{1,m}$, $\text{Stored-PERSONNEL}_{2,m}$,
   $\text{Stored-PERSONNEL}_{3,o}$, and $\text{Stored-PERSONNEL}_{4,o}$.

3. A third consisting of

   $\text{Stored-PERSONNEL}_{1,m}$, $\text{Stored-PERSONNEL}_{2,o}$,
   $\text{Stored-PERSONNEL}_{3,n}$, and $\text{Stored-PERSONNEL}_{4,o}$; and

4. Finally, one consisting of

   $\text{Stored-PERSONNEL}_{1,m}$, $\text{Stored-PERSONNEL}_{2,o}$,
   $\text{Stored-PERSONNEL}_{3,o}$, and $\text{Stored-PERSONNEL}_{4,o}$.

Similarly there are six ways of reconstructing the INVENTORY relation in that example.

A collection of stored fragments which form a complete and non-redundant copy of the logical database is called a materialization. At any given time SDD-1 recognizes a specific set of materializations as "supported". A user logging in to SDD-1 is given access to the database through a specific supported materialization and repeated accesses to the system will result in assignment to the same supported materialization.

The make-up of each supported materialization is recorded in a table such as the one illustrated in Figure 2.5. The management of this and other directory information will be described in Section 2.1.4. Another table indicates the

Table of Fragment Assignments Figure 2.5

Supported Materialization	PERSONNEL relation fragments 1	2	3	4	INVENTORY relation fragments 1	2	3
1	m	m	n	o	m	m	o
2	m	m	o	o	o	n	o
3	m	o	n	o	m	m	o
4	m	o	n	o	n	n	o
5	m	o	o	o	m	m	o
6	m	o	o	o	n	n	o

Table entry is datamodule from which the indicated materialization obtains each fragment. Note that the table shows supported materializations only and does not show all possible materializations.

materialization to which each user is assigned. SDD-1 will service a user's retrieval requests by accessing only the stored fragments listed for his assigned materialization.

The concept of supported materializations (hereafter called simply "materializations") is an important element of SDD-1's data distribution model. Since each materialization is a complete and non-redundant copy of the database, it is analogous to the simpler concept of logical relations in the non-distributed database setting. The materialization concept plays a central role in the notion of database consistency in SDD-1 and has a major

impact on the problem of updating redundantly stored data.

This issue is addressed in Section 2.2.

2.1.4 Directory Management

One issue which is frequently cited (e.g., in [FRY and SIBLEY], [SIBLEY], [STONEBRAKER and NEUHOLD]) as significant in the design of distributed database systems is the management of the global system directory. This directory provides the information which the GDM's require to parse user requests, to locate data of interest, and to choose accessing and updating strategies.

Alternatives for handling the directory include:

- storing it in a single, central datamodule;
- storing a complete copy at each datamodule;
- dispersing the directory over the set of datamodules non-redundantly; and
- dispersing the directory over the set of datamodules with some redundancy.

The optimal choice among these alternatives depends upon the pattern of transaction traffic in the network. Ideally, general purpose distributed database systems would not adopt any of these alternatives directly but rather would permit the choice to be made as part of database design.

SDD-1 achieves this flexibility by treating the directory as ordinary user data. The data which comprise the directory are partitioned into fragments just as all other user data are, and like user data, these fragments are stored in a distributed and redundant manner throughout the system. In this way arbitrary levels of redundancy and distribution of the directory can be supported. It should be noted that treating the directory as user data also makes other system features such as security, integrity, and concurrency control available for directory management.

There is one regard in which directory information cannot be treated identically to user data and that is in the matter of where the directory information for the directory itself is stored. It is necessary to treat this information specially in order to provide the system with a known starting point from which directory information can be found.

This problem is handled in SDD-1 by factoring the directory information for directories into separate relations which are stored in every datamodule. The decision to store this level of directory information at all datamodules is based on the assumption that these relations are small and very retrieval intensive.

2.2 Redundant Updating

The focus of the redundant update problem is to develop techniques for updating redundantly stored data in a manner that

a. preserves database consistency, and
b. avoids excessive inter-computer synchronization overhead.

This section states criteria for database consistency so as to clarify the property that the redundant update algorithm must preserve. This section also discusses previous solutions to the problem, explaining why these previous solutions are inadequate.

In a redundant database system the notion of database consistency has two aspects:

1. Mutual consistency of the redundant copies; and

2. Internal consistency of each copy.

The mutual consistency criterion states that all copies of the database must in fact be identical copies of each other and not diverge over time. Though it is not possible for the copies to be instantaneously identical at all times, they must attain the same final state were all database activity to cease.

The internal consistency criterion specifies that each distinct copy of the database must remain consistent within itself just as a conventional non-redundant database must. Our definition of internal consistency is based on the concept of serializability ([ESWARAN et al], [GRAY et al], [HEWITT]). Serializability dictates that the effect of a set of concurrent transactions on a database must be equivalent to some serial, non-overlapping sequence of those same transactions. Then if each transaction running seperately would not violate database consistency, the effect of the concurrent collection cannot be inconsistent.(1)

Conventional centralized database systems ensure serializability through locking mechanisms ([ESWARAN et al], [GRAY et al], [CHAMBERLIN et al a,b]). It is

---

(1) Verifying that seperate transaction will not violate consistency is an integrity checking step of the sort described by [HAMMER and MACLEOD]. It will not be considered here.

possible to extend these locking mechanisms to the distributed environment, however distributed locking is inefficient. A straightforward distributed locking algorithm (cf [ROTHNIE and GOODMAN]) requires 5n inter-computer messages to propagate an update to n datamodules:

> n lock request messages, n lock grant messages, n messages to transmit the update, n update acknowledgements, and n messages to release the locks.

Furthermore, the delay in executing the update is lengthy. It is:

the maximum delay encountered in setting the locks
+
the maximum delay encountered in performing the update

For these reasons the straightforward locking approach is inadequate for SDD-1 and other methods must be sought.

[THOMAS- b] has described a solution that employs a "voting" protocol to enable lock setting by a majority of datamodules rather that requiring unanimous approval. The method reduces the cost of the database locking by about a factor of two in terms of total inter-computer communication required.

However, even with this reduction a large amount of communication is needed to perform updates. It appears that this method will perform unacceptably in networks containing large numbers of datamodules.

A rather different approach has been proposed by [ALSBERG and DAY]. This approach requires that all update activity in the database be funneled through a single datamodule called the primary site. All locking operations are performed within that single datamodule and therefore no additional synchronization communication is required.

The primary site approach avoids excessive synchronization communication but at a cost of re-introducing three important drawbacks of conventional centralized systems:

- The responsiveness of the system to updates is no better than that of a centralized system located at the primary site;
- As with a centralized system one must upgrade datamodules that are already in place in order to increase system capacity -- it is not possible to add datamodules incrementally;

- If the transaction is initiated at a significant distance from the primary site high communication cost or long communication delay is incurred in transmitting an update to the primary site.

Both of these previous solutions have drawbacks that diminish their effectiveness in a general system for distributed data. Consequently we have sought to develop new techniques for SDD-1. The remainder of this section presents the redundant update methodology of SDD-1.

2.2.1 Protocols and Protocol Selection

In this section we describe more precisely the procedures used by each datamodule to synchronize and exchange information with other datamodules and the machinery used to choose the particular procedure to employ in handling a given transaction.

2.2.1.1 Primitive Actions

There are two primitive actions that a datamodule can perform which modify the state of the database.

The first of these is the local execution of a transaction. This action involves the following steps:

1. a transaction is introduced at a datamodule;
2. the input data to the transaction is read from the local copy of the database;
3. the function called for by the transaction is applied to this data; and
4. the result is written into the local copy of the database.

This primitive action is denoted $L_{t,m}$ to indicate the local processing of transaction t at datamodule m.

When an update transaction has been fully executed locally, the datamodule involved transmits the database changes induced by that transaction to all other datamodules so that each can make the changes in its own database copy. The processing of this update message at the receiving module is the second primitive action. It

is denoted $U_{t,n}$ to indicate the processing of the update message associated with transaction t at datamodule n.

It is important to note that both of these primitive actions are assumed to be atomic. Local locking at each datamodule is used to enforce this assumption.

In general, a given data item may be updated from two or more datamodules and the update messages associated with these transactions may arrive at other modules in arbitrary orders. To ensure that all copies of the database converge to the same state, all update messages, $U_{t,n}$, are time stamped with the time at which $L_{t,m}$ ran according to the clock in datamodule m. (If appropriate disciplines are employed in handling the clocks in each datamodule, it is not important that the clocks be synchronized. See [LAMPORT], [THOMAS-b].) This time stamp is denoted TS(t). When datamodule n modifies a data item in response to $U_{t,n}$ it marks that data item with the time stamp TS(t). Each datamodule i will then employ the following rule in processing $U_{t1,i}$:

> Change the local copy of a data item designated by $U_{t1,i}$ if and only if the time stamp of that data item is less recent than TS(t1).

It can easily be shown that this discipline is sufficient to ensure that all database copies converge to the same value (consistency condition #1). This scheme is comparable to the use of time stamps by [THOMAS b].

However the time stamping rule is not adequate to ensure the internal consistency of each copy (condition #2). Section 2.2.1.2 describes the additional synchronization and update ordering mechanisms necessary to achieve that goal.

2.2.1.2 Protocols

The intermodule coordination procedures described here are called protocols. There are three protocols defined for use in the system. The particular protocol to be used in processing a given transaction is determined by a protocol selector function, $S_m(t)$, which indicates what protocol should be used for processing transaction t introduced in datamodule m.

The set of selector functions used for all datamodules in the system is called the protocol configuration and it is established during database design. Section 2.2.2 will

describe the conditions that a protocol configuration must obey to ensure that all possible transactions run according to its selector functions result in a consistent database.

The three protocols incur substantially different delays in handling transactions. In establishing a protocol configuration, the designer attempts to ensure that most transactions may use the quickest protocol, a smaller number use the medium speed protocol, and a smaller number still require use of the slowest one.

The three protocols operate as follows:

Protocol 1 -

Datamodule j processes update messages from datamodule i in the order they were sent from i. That is, for every pair of datamodules (i,j), the update messages $U_{t,j}$ resulting from local executions in module i (i.e. actions of the form $L_{t,i}$), will be processed in j in the same order as the local executions in i. (This is the same as the time stamp ordering of these messages.)

This is an easy condition to enforce in an Arpanet-like communications environment ([METCALFE)] since all messages are delivered in the order sent.

Protocol 2 -

All update messages arriving at j are processed in time stamp order regardless of the datamodule of origin. The effect of this condition can be achieved by the following transaction processing rule. Prior to performing the local execution of t, (i.e. $L_{t,j}$) datamodule j will examine the time stamps of all data items read by t and choose the most recent one. Call this time stamp r. If there exists some materialization i which has not sent a message to j with a time stamp as recent as r then j will wait for such a message before executing t. This wait can be made arbitrarily short by requiring every module to send a null message to every other module every s seconds.

Protocol 3 -

This protocol is equivalent to global locking and it is clearly the most expensive procedure. When datamodule j wishes to execute t using this protocol it first sends a message to every other datamodule requesting a lock on the data items which t will read or write. These locks are called g-locks and their effect is to prevent a transaction at datamodule i from reading anything which t writes or writing anything which t reads until the g-lock has been cleared. (This request could of course lead to deadlock, but the usual deadlock resolution schemes can handle such situations.)

When module i receives a g-lock and sets it locally, it sends an acknowledgement back to j in the same logical communication stream as its update messages. This ensures that datamodule i will process all update messages $U_{t,'j}$ with time stamps more recent than the lock acknowledgement.

Next datamodule i processes $L_{t,j}$ and finally sends out the associated update messages and lock releases.

This protocol can incur a very substantial delay. A major objective of the technique being described here is to minimize the use of this procedure.

2.2.1.3 Protocol Selection

The concept of a protocol selection function, $S_m(t)$, was introduced above. It is a function that determines which protocol should be used in executing transaction t in datamodule m. The operation of the function is driven by a table which is established as a step in database design. Section 4 describes the conditions this table must satisfy to ensure that the database will remain consistent in the presence of update transactions running under the protocols which this table designates.

In this section, we describe the operation of the protocol selector function given a correct table. That is, we address the question, "How is a protocol chosen when a transaction t is introduced at datamodule m?"

Protocol selector functions are based on the concept of transaction classes. A transaction class is a set of transactions characterized by the set of data items that transactions in the class read and the set of data items they write. A class, C_i, is characterized by two functions:

$R\$(C_i)$ - the set of data items read by transactions in the class (called the read set); and
$W\$(C_i)$ - the set of data items written by transactions in the class (called the write set).

C_i is the set of transactions whose members all read exclusively from the class read set and write exclusively into the write set. Formally:

$$C_i = \{T/R(T) \quad R(C_i)SW(T) \quad W(C_i)\}$$

Read and write sets are defined by simple predicates.

The table which drives the protocol selector function defines a function which maps each class into one of the

three protocols. The function is denoted $P(C_{i,m})$ and yields the protocol number to be used for transactions introduced in module m which are members of class $C_{i,m}$. A transaction which is not a member of any class is mapped into protocol 3. In general a transaction may be a member of several classes and in these cases the selector function chooses the class with the lowest protocol number. Formally:

$$S_m(t) = \min \{P(C_{i,m}) \;/\; t \;\; C_{i,m}\}$$

2.2.2 Safe Protocol Configurations

Section 2.2.1 described the protocols available for synchronizing transactions and transmitting updates among datamodules. The objective of these protocols is to preserve database consistency while performing updates in an efficient manner. The correctness of this interaction, in terms of the consistency of the database, depends on which transactions are processed using which protocol. This section discusses the nature of protocol configurations which will always yield consistent databases. Such configurations are termed safe protocol configurations.

2.2.2.1 Global Logs and Consistency

The central framework of the safety analysis involves an abstraction known as the global log. The construction of the global log is described in 2.2.2.2. In the present discussion we will simply define this object and use it to examine database consistency.

The global log is a sequence of the system primitive actions (local transaction executions and the processing of update messages) defined in section 2.2.1.1. Figure 2.6 shows an example of a global log involving 5

A Global Log Figure 2.6

$L_{1,1}\ L_{2,3}\ L_{3,2}\ U_{1,2}\ L_{4,3}\ U_{3,1}\ U_{4,2}\ U_{4,1}\ U_{2,1}\ U_{3,3}\ U_{1,3}\ U_{2,2}\ L_{5,1}\ U_{5,3}\ U_{5,2}$

---

transactions and 3 data modules. The global log should be viewed as an actual sequence of steps which could be taken by the system. The global log is an abstraction which serializes events in the system which may in fact have occurred concurrently.

Another central concept in this analysis is the idea of a database state. Informally the state of the database is the collection of all data item values at all datamodules. Formally it is the collection of pairs of the form (i, $state_i$) where i is the designator for a data module and state is the collection of relations represented at i.

A primitive action L or U may change the value of a data item and therefore change the state of the database. The execution of a global log, a sequence of primitive actions, may therefore change database state. We say that a global log G_i, defines a function g_i, which maps one database state, DS_1, into another state DS_2. That is

$$g_1 (DS_1) = DS_2.$$

If two global logs define the same function then the logs are said to be equivalent. That is:

if DS_i $g_1(DS_i) = g_2(DS_i)$
then G_1 and G_2 are equivalent

Section 2.2.2.3 defines a set of rules for transforming logs into equivalent logs. Each of these rules will indicate when two adjacent actions in G may be permuted without affecting the function g. For example, two L's which do not read or write anything in common may always be switched without changing the effect of the log on database state.

Consistency condition #2 states that for a database state to be consistent, it must be possible to produce that state by a serial sequence of the transactions which have been processed by the system. This condition may be stated in terms of global logs:

> Let DS_2 be the current state and assume DS is an initial state which is known to be consistent. Then DS_2 is consistent if there exists a global log, G_s, such that $g_s(DS_1)\$=\$DS_2$ and each $L_{t,i}$ in G_s is immediately followed by all of its update messages $U_{t,j}$.

Such a global log is called serial. Figure 2.7 shows an

A Serial Global Log Figure 2.7

$$L_{1,1}\ U_{1,2}\ U_{1,3}\ L_{3,2}\ U_{3,3}\ U_{3,1}\ L_{2,3}\ U_{2,1}\ U_{2,2}\ L_{5,1}\ U_{5,2}\ U_{5,3}\ L_{4,3}\ U_{4,1}\ U_{4,2}$$

---

example of a serial global log. A global log which is not serial but which is equivalent to a serial log is called serially reproducible (SR).

It follows that:

if DS_1 is consistent and

$DS_2 = g(DS_1)$ and

G is SR

then DS_2 is consistent.

This suggests a procedure for determining whether a database state is consistent: first find a global log which could have produced that state from a known consistent starting point, and then show that the log is SR. Section 2.2.2.2 addresses the problem of constructing global logs, and then 2.2.2.3 examines mechanisms for transforming logs into equivalent serial ones.

2.2.2.2 Constructing a Global Log

Global logs are constructed in two steps. First a local log is constructed for each datamodule by recording the actual sequence in which actions are performed at the datamodule. Then the local logs are combined into a global log in such a way that the effect of the global log on each datamodule is the same as that of the original local log.

This aggregation can be accomplished by a merge of the local logs which obeys the following constraints:

1. The ordering of actions from each local log is preserved. That is, the combination is a merge.
2. L actions from different local logs are placed in time stamp order.
3. All of the update messages, $U_{t,j}$ from transaction t appear in the log after $L_{t,i}$.

It can be shown that a global log constructed in this way will produce precisely the same database state in each datamodule as that produced by the original local logs.

2.2.2.3 Global Log Transformation Rules

In this section we examine a family of functions each member of which will map a global log into an equivalent global log. These functions are of the form:

$$E_i\ (G_2) = G_2$$

The effect of E_i is to permute the i^{th} and $i + 1^{th}$ members of the log as long as the log which results from the switch is equivalent to G_1. If it is not, then the switch is not performed. We can define E_i in terms of another function S_i which will permute the i^{th} and $i + 1^{th}$ terms unconditionally.

$$E_i(G_1) = S_i(G_1) \text{ if } S_i(G_1) \text{ is equivalent to } G_1$$
$$= G_1 \text{ otherwise}$$

Under what circumstances will the application of S_i, the permutation of two adjacent actions, produce a non-equivalent log? Table 2.1 shows those situations in which a switch may produce a global log with a different effect on database state from the original. In general non-equivalence depends on the types of actions involved (L or U), on whether the actions are taking place at the same datamodule, and on the intersections of read and write sets.

Given these transformation rules we are now prepared to consider when a log is SR.

Global Log Transformation Rules Table 2.1

Each row in the table represents a possible i^{th} entry in the log and each column represents a possible $i+1^{th}$ entry. The contents of each square indicates when the entries may be interchanged to produce an equivalent log.

The meaning of each entry is as follows:

blank - switch can always be made

never - switch can never be made

X - sequence is not possible in a well-formed log

R-W - switch cannot be made if

$R(t_i) \cap W(t_{i+1}) \neq \phi$ (i.e the entries cannot be interchanged if the read set of the action at i intersects with the write set of the action at i + 1.)

W-W - switch cannot be made if

$W(t_i) \cap W(t_{i+1}) \neq \phi$

W-R - switch cannot be made if

$W(t_i) \cap R(t_{i+1}) \neq \phi$

When more than one entry appears in a square either condition prevents a switch.

Global Log Transformation Rules Table 2.1

i + 1

i	$L_{t1,i}$	$L_{t2,i}$	$L_{t3,j}$	$U_{t1,j}$	$U_{t2,j}$	$U_{t3,i}$
$L_{t1,i}$		R - W W - W W - R	W - W	Never		R - W
$L_{t2,i}$	R - W W - W W - R		W - W		Never	R - W
$L_{t3,j}$	W - W	W - W		R - W	R - W	Never
$U_{t1,j}$			W - R			
$U_{t2,j}$			W - R			
$U_{t3,i}$	W - R	W - R				

2.2.2.4 Serially Reproducible Global Logs

A global log is said to be serially reproducible (SR) if it is equivalent to a serial log. An SR global log can never violate the consistency of a database.

In this section we consider a technique for determining whether an arbitrary log is SR. This technique involves the repeated application of the E_i functions in an algorithm which attempts to transform a global log into some equivalent serial log. This algorithm is called ES and it maps a log G into a serial log G if a serial log exists: $ES(G) = G_s$. The details of this algorithm will not be described here. However, one key step in its operation will be discussed.

In computing the ES function, one reaches a stage in which the algorithm is attempting to move a particular $U_{t,j}$ to a position adjacent to $L_{t,i}$ by the repeated application of E_i.

The strategy for accomplishing this is to permute $U_{t,j}$ and its immediate predecessor until the L and U are adjacent. At some point in this process E_i may not be

able to switch $U_{t,j}$ and its predecessor because doing so would lead to a non-equivalent log.

If this circumstance arises then the strategy for moving L and U together must be modified. Suppose that the action which immediately precedes $U_{t,j}$ and which cannot be switched with $U_{t,j}$ is A. Such an action is called a blocking action. ES now attempts to move A past $L_{t,i}$ so that it can no longer block $U_{t,j}$. In doing so, A can also be blocked and then the process repeats itself. Eventually ES encounters either a blocking action which can be moved past $L_{t,i}$ or discovers that $L_{t,i}$ itself is a blocking action. In the latter case no serial log equivalent to the starting log G exists. Hence G is not SR.

We have developed a graphic technique which clearly identifies non-SR logs. Given a global log G we construct a graph by drawing an arc between every pair of actions which, if adjacent, could not be interchanged. This graph is called a global log graph. Table 2.1 can be viewed as a set of rules for drawing these arcs. Figure 2.8 shows an example of a log with these arcs added.

The paths which appear in the global log graph are the key to the SR question. Returning now to the ES algorithm and the problem of moving $L_{t,i}$ and $U_{t,j}$ together, the following result can be stated:

A Global Log Graph Figure 2.8

The graph depicts three transactions:

1. t1 run locally at datamodule 1
2. t2 run locally at datamodule 2
3. t3 run locally at datamodule 3

The read and write sets of the transactions intersect as follows:

$$W(t1) \cap W(t2) \neq \phi$$
$$W(t1) \cap R(t3) \neq \phi$$
$$W(t2) \cap R(t3) \neq \phi$$

The Global Log Graph for these transactions is:

$$G = L_{1,1}\ L_{2,2}\ U_{2,3}\ L_{3,3}\ U_{1,3}$$

G is not SR since $L_{t,i}$ and $U_{t,j}$ cannot be made adjacent.

If $L_{t,i}$ and $U_{t,j}$ cannot be made adjacent through the repeated application of the E_i function, then there exists a path from $L_{t,i}$ to $U_{t,j}$ whose nodes are a non-empty subset of the actions between $L_{t,i}$ and $U_{t,j}$.

This result provides a means of recognizing that a global log is not SR and represents an important piece of a broader goal -- ensuring that all possible global logs

produced by a given set of transactions are SR. If the broader goal is achieved then we have the means for recognizing in advance that the set of transactions can never lead to an inconsistent database. Section 2.2.2.5 addresses this topic.

2.2.2.5 L-U Graph Analysis

Two issues determine whether a given set of transactions will always produce an SR global log. The first issue is the nature of the intersections among the read and write sets of the transactions. In this section we introduce a graphic technique called L-U graphs which capture the relevant aspects of these intersections.

L-U graphs represent the paths in a global log graph but drop the ordering information inherent in a particular global log. In addition, L-U graphs do not require knowledge of the particular transactions running in the system; they require only the definition of classes of transactions which may be running.

The second issue is the protocol employed for each transaction. The protocols introduce controls on the interleaving of transactions which limit the types of

local logs which may appear and hence limit the types of global logs as well.

At the conclusion of this section a series of results are presented which combine the two issues. These results lead directly to the specification of safe protocol configurations.

An L-U Graph Figure 2.9

Classes: $C_{1,1}$, $C_{2,1}$, $C_{3,2}$, $C_{4,3}$

$W(C_{1,1}) \cap R(C_{2,1}) \neq \phi$
$W(C_{1,1}) \cap R(C_{3,2}) \neq \phi$
$W(C_{3,2}) \cap W(C_{4,3}) \neq \phi$

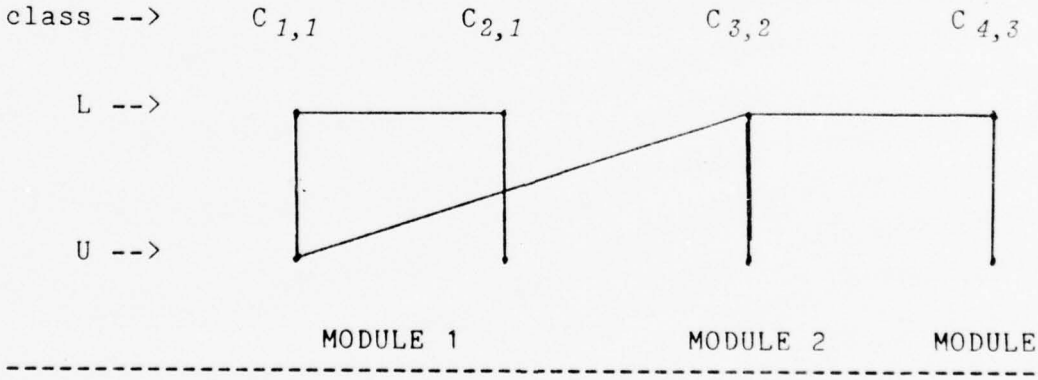


Figure 2.9 shows an example of an L-U graph. The graph contains a pair of nodes, labeled L and U, for each defined transaction class as discussed in Section 2.2.1.3. The L node represents the local execution action of

transactions in the class and the U node represents the update message actions of the transactions. By convention we always draw the L node for a class immediately above its U node and always group together all classes of a given datamodule.

Arcs are drawn between the following pairs of nodes:

1. L and U for the same class (called vertical arcs);
2. L and L for classes at the same datamodule if there is an R-W, W-W, or W-R intersect between the classes (called horizontal arcs);
3. L and L for classes at different datamodules if there is a W-W intersect (called horizontal arcs); and
4. L and U for classes at different datamodules if there is an R-W intersect (called diagonal arcs).

Given an L-U Graph, a safe protocol configuration is constructed by considering each class $C_{m,i}$ in turn and applying the following rules regarding the graph topology:

1. If neither node of $C_{m,i}$ is on a cycle then the protocol for transactions in $C_{m,i}$ is 1.

$$P(C_{m,i}) = 1$$

2. If either node of $C_{m,i}$ is on a cycle then apply the following topological test to the arcs of the cycle which impinge on $C_{m,i}$:

   a. if graph is of the form

      $P(C_{m,i}) = 1$

   b. if graph is of the form

      $P(C_{m,i}) = 2$

   c. if graph is of the form

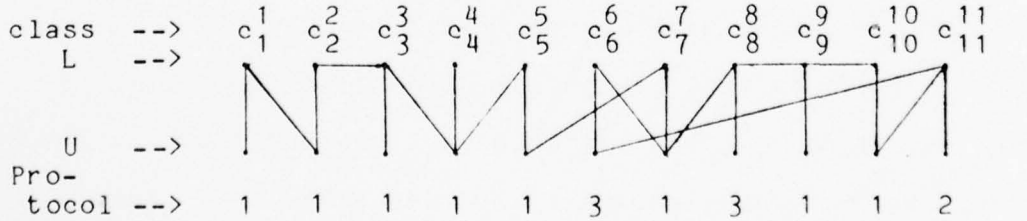
      $P(C_{m,i}) = 3$

   If a node of $C_{m,i}$ *appears on more than* one cycle then the highest numbered protocol is used.

Figure 2.10 shows an example of an L-U graph with labeling to indicate which protocol to use under the rules

---

An L-U Graph with Protocols Indicated Figure 4.5



---

described above. The correct operation of a system based on this protocol selection scheme has been proven using global log analysis by [BERNSTEIN et al].

The degree of effectiveness of a system based on the redundant updating concepts described here depends upon the relative frequency of transactions requiring each of the protocols. We speculate that for most applications it is possible to construct safe protocol configurations in which most transactions run under protocol 1, a smaller number under 2, and still fewer under 3.

2.3 Dispersed Data Access

The method described here for the retrieval of dispersed data always operates in the context of a specified materialization, and thus from its point of view the database appears to be distributed but non-redundant. That the database is, in fact, redundant is of no concern to our retrieval technique and hereafter will not be considered. (The methodology used by SDD-1 to maintain redundant database copies is described in section 2.2, [ROTHNIE et al] and [BERNSTEIN et al].)

A central constraint on our retrieval algorithm is that communication between database sites in SDD-1 is via relatively slow channels. Using the Arpanet as the archtypical communication channel in SDD-1, the bandwidth between sites is some 50-100 times slower than the bandwidth between each computer and its local disk storage. Consequently the minimization of data transfer from site to site is the primary objective in optimization though not the only one.

The retrieval method is based on a number of assumptions which can be stated as follows:

a. For the duration of any one query, the data model remains fixed and is a relational model. Since the method operates in the context of one materialization, there are no redundantly stored relations in the model. For expository convenience we assume that each relation resides at a single site; there is no loss of generality in this assumption and the extension to relations fragmented among several sites is straightforward.

b. Only fragments of relations in the data model are moved from one site to another, a fragment being a subrelation, a projection, or a combination of these two operations.

c. It is a matter of indifference as to where the final result is produced, as long as it is produced at a single site.

Given a query which references data distributed over several sites, our goal is to reduce it to a sequence of local queries with data transfer between local processing steps, and to do it in a way so as to minimize the combined communication and processing cost.

To fix ideas, let us consider a specific example to which we shall refer throughout the remainder of this paper:

Example 2.1

Relation (Domains)	Site	Variable in Query
Supplier (S#, City)	A	S
Availability (S#, P#, QOH)	A	V
Parts (P#, Pname)	B	P
Projects (J#, City)	C	J
Supply (J#, S#, P#, Qty)	C	Y

Query

```
RETRIEVE (S.S#)
WHERE (S.S# = V.S#) AND (S.City = J.City) AND (S.S# = Y.S#)
  AND (V.P# = P.P#) AND (V.QOH > Y.Qty) AND (P.P# = Y.P#)
  AND (P.Pname = 'Bolts') AND (J.J# = Y.J#)
  AND (Y.Qty > 1000)
```

Stated verbally, the query asks for the S# for those suppliers who supply bolts to projects located in the same city in quantity greater than 1000 and for whom the quantity-on-hand is larger than the quantity supplied. It is useful to have a graphical representation for the query as is done in Figure 2.11 where lines represent linking terms.

2.3.1 A Difference of View

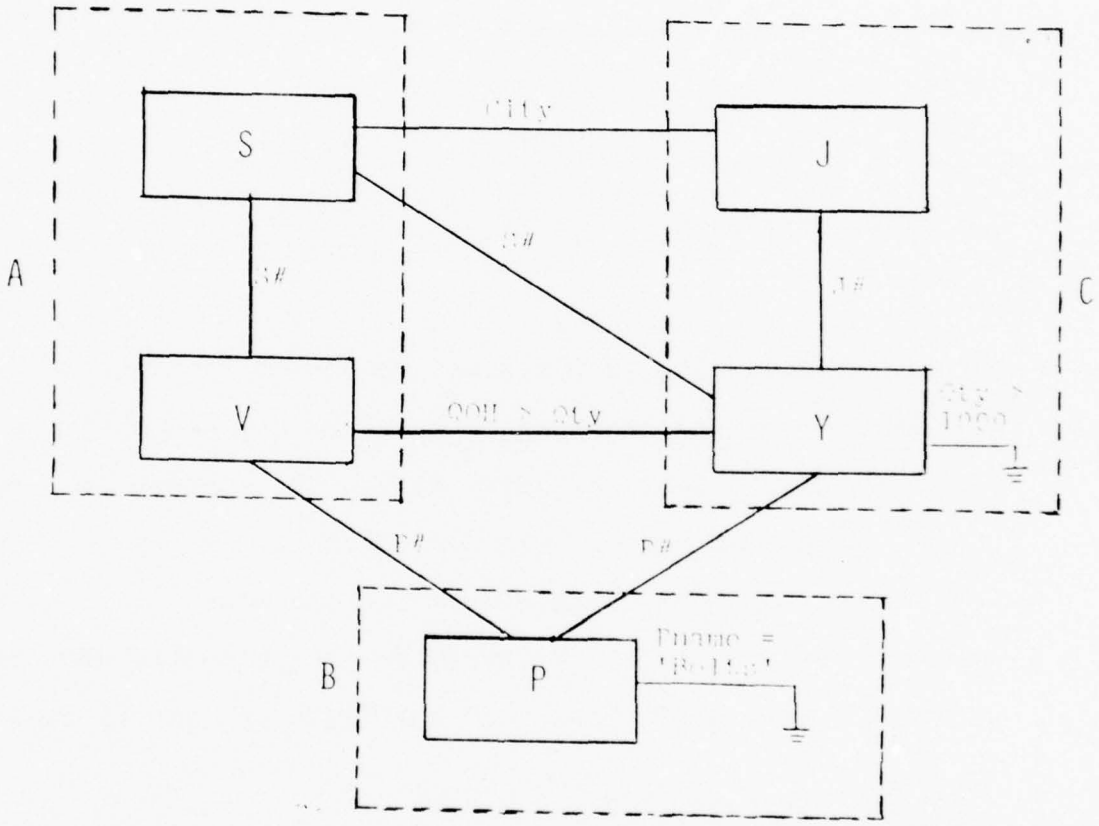
The added dimension in dispersed retrieval as compared to retrieval from a centralized database is the necessity to transfer data between database sites. As a first step in query processing it seems desirable to separate the complexity inherent in the query from the added difficulty due to distribution. This can be rather neatly done by adopting a series of "views" of the database as follows:

V_1: The User View

This consists of the data model, plus possibly some or all of the distribution information.

V_2: The Distribution View

Graphical Representation of Example 2.1 Figure 2.11



This view consists of a collection of relations, one per site, each being the cartesian product of all the relations at one site. For the example in section 2.3 V_2 consists of the following relations:

Example 2.2

RELA (SS#, SCity, VS#, VP#, VQOH)
RELB (PP#, PPname)
RELC (JJ#, JCity, YJ#, YS#, YP#, YQty)

Our convention in renaming is self-explanatory.

V_3: The Local Views

There is one local view per site, consisting of all the relations of the data model at that site.

We observe that in the distribution view (V_2) the most prominent features are precisely the boundaries of data dispersion. Expressed in V_2, a query gives explicit display to its multilocational nature. A query is local (i.e., of single site) if and only if its V_2 expression is one-variable. This suggests that the concept of decomposition (reducing a multivariable query to a sequence of one-variable queries) (cf [STONEBRAKER et al], [WONG and YOUSSEFI]) be adapted to reduce a distributed query to a series of local queries.

The overall plan of query processing that we propose can be separated into the following steps:

1. conversion V_1 query --> V_2 query
2. distribution multivariable V_2 query --> one-variable V_2 queries
3. reconversion one-variable V_2 query --> multivariable V_3 query
4. decomposition multi-variable V_3 query --> one-variable V_3 queries
5. local one-variable query processing

Conversion and reconversion (steps 1 and 3) are both mechanical and fast so that nothing further about them needs to be said. Step 4 does not involve the distributed nature of the database in any way. While the optimization problem of decomposition is by no means completely solved, the focus of our attention in this paper is on the distributed nature of the database, and purely local optimization problems will not be considered further. What remains then is the problem of optimizing the reduction of a distributed query into local queries (step 2), a procedure that we call distribution.

Example 2.3 Suppose that we consider the query in section 2.3 to be a V_1 query. The corresponding V_2 query takes on the form:

```
RANGE OF (A, B, C) IS (RELA, RELB, RELC)

RETRIEVE (A.SS#)
WHERE (A.SS# = A.VS#) AND (A.SCity = C.JCity)
  AND (A.SS# = C.YS#) AND (A.VP# = B.PP#)
  AND (A.VQOH > C.YQty) AND (B.PP# = C.YP#)
  AND (B.PPname = 'Bolts') AND (C.JJ# = C.YJ#)
  AND (C.YQty > 1000)
```

2.3.2 The Basic Tactics of Distribution

The basic tactics proposed in [WONG and YOUSSEFI] to decompose a multivariable query into one-variable queries are the following:

a. One-Variable Subqueries: Portions of the query may entail reducing single relations by restriction and/or projection. These one-variable operations should always be performed first.

b. Reduction: When a query can be separated into two parts with only a single overlapping variable, it is often advantageous to do so.

c. Tuple Substitution: One of the variables is successively replaced by the actual tuples in the range relation of the variable.

Since distribution is decomposition in V_2, it is of interest to interpret these tactics in terms of the distributed database.

a. A V_2 one-variable query is a local query. It seems clear that what can be processed locally to reduce the sizes of the relations should always be done first. Hence, one-variable subquery processing should be retained as a tactic in distribution.

b. Interpreted in terms of the distributed database, reduction means a partition of the network into subsets which overlap at a single site. This could be advantageous, but the algorithm that we shall present makes no use of this tactic.

c. Tuple substitution in V_2 implies the transfer of data one V_2-tuple at a time between sites. However, bulk transfer of data is undoubtedly more efficient than tuple-at-a-time transfers. Therefore if a variable is to be tuple-substituted, it is often better to move the entire relation.

There are situations where not every tuple needs to be substituted, but on balance we decide in favor of moving relations rather than tuples. Consequently we replace tuple-substitution as a tactic by:

Move relation R from site A to site B

Observe that the application of the "move" tactic is restricted, by assumption, to only fragments of the relations in the data model. The role played by "move" in distribution is quite similar to that of "tuple-substitute" in decomposition, each being a necessary but undesirable operation. Like tuple-substitution in decomposition, repeated "moves" will completely distribute any query. However, the object of optimization in distribution will be to move as little as possible.

2.3.3 A Strategy of Moves

A distribution strategy consists of a series of move relation tactics with local processing between them. Since communication costs are assumed to be high relative to local processing, the optimization of a distribution strategy depends principally on the optimization of its move tactics, and the optimization of moves can be undertaken largely independently of local processing optimization. We make use of this independence in designing our algorithms, although the cost of local processing is not entirely ignored.

For any query Q, one strategy that is always possible is the following:

a. Do all the local processing that can be done with no data transfer at all.

b. Then, make the minimum moves which would render the residual V_2-query one-variable without further local processing.

In such a strategy the moves are all made in parallel without intervening local processing. For this reason, it is in general far from optimal. However, it does represent an initial strategy that we can refine in successive steps.

Let M(Q) denote the set of moves defined by (b). Each element in M(Q) is of the form: m = "Move R from A to B".

Example 2.4 For the query in our example, define

R_1 = (Parts {Pname = 'Bolts'}) [P#]

R_2 = (Supplier (S# = S#)
(Availability {QOH > 1000}))
[S#, City]

R_3 = (Supplier (S# = S#)
(Availability {QOH > 1000}))
[S#, P#, QOH]

where R[A] denotes the projection of R on domain A, R{predicates} denotes the restriction of R defined by the predicate, and R(A = B)S the equijoin of R and S on A and B.

One possible value for M(Q) is the following:

$$M(Q) = \left\{ \begin{array}{l} m_1 = \text{"move } R_1 \text{ from B to C"} \\ m_2 = \text{"move } R_2 \text{ from A to C"} \\ m_3 = \text{"move } R_3 \text{ from A to C"} \end{array} \right\}$$

The actual value of M(Q) would depend on the sizes of the various relations.

The general plan now is to replace M(Q) by two sets of moves, M_1 and M_2, that are to be executed sequentially with local processing between them. The objective in selecting M_1 and M_2 is for the total cost of the moves together with the cost of any additional local processing to be less than the cost of M.

Example 2.5 M(Q) is the same as before.

$M_1 =$ $\{m_4 =$ "move R_1 from B to A"$\}$

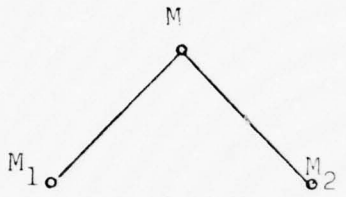
Define R_2' and R_3' by the same formulas as R_2 and R_3 respectively, except that the relation "Availability" is replaced by its join with R_1. The construction of this join constitutes the local processing between M_1 and M_2. Now set

$$M_2 = \left\{ \begin{array}{l} m_5 = \text{"move } R_2' \text{ from A to C"} \\ m_6 = \text{"move } R_3' \text{ from A to C"} \end{array} \right\}$$

Observe that since $R_3'[P\#] = R_1$ there is no need to include "move R_1 from A to C" in M_2.

Because $R_2' \subset R_2$ and $R_3' \subset R_3$, $M_1 + M_2$ clearly entails moving less than M(Q) if we assume that moving R_1 from B

to A costs no more than moving it from B to C. We can represent the transformation M(Q) --> (M_1 followed by M_2) graphically as follows:



where it is understood that the left offspring always precedes the right with local processing in between.

It is possible that we can continue to split the nodes, and in general end up with a binary tree where the leaf nodes represent all the moves that are to be undertaken. The moves within one leaf can be made in parallel and the leaves are executed sequentially from left to right. The criterion for node splitting is as follows:

A node N is split into N_1 and N_2 if and only if

a. the combined cost of N_1 and N_2 is less than N alone, and

b. the pair (N_1, N_2) is minimum in cost among all pairs satisfying (a).

The resulting algorithm might be termed a "greedy" algorithm. In general it cannot be optimum, since it can never climb a hill surrounded by valleys, nor is it guaranteed to seek the highest peak if there are more than one. The algorithm could, in theory, be expanded by permitting condition (a) to attempt more global minimization. However, such an extension greatly increases the number of candidate splits which would have to be considered.

In the overall scheme that we have outlined, any relation which is the result of local operations is a candidate to be moved. Potentially, the number of such relations is enormous. The strategy that we have presented is designed to: (a) severely limit the number of such candidates that need to be considered, and (b) make the optimization recursive so that the overall algorithm can be expressed in terms of what is to be done at a single step.

There are only two basic ingredients in the algorithm: determination of M(Q) and splitting a node in the strategy tree. In section 2.3.4 we explain how M(Q) is determined; section 2.3.5 then explores node splitting.

2.3.4 Determination of M(Q)

As described in section 2.3.3, M(Q) represents a first cut at obtaining a good distribution strategy for processing a query Q. M(Q) is obtained by limiting the local processing to operations that can be performed in the database prior to any moves of M(Q).

To determine M(Q) we first determine what local processing is possible before any moves. The effect of this local processing is to reduce the sizes of the fragments that must be moved. Then we compute the cost of moving all the reduced size fragments to a given destination site S_{DEST} for all posible values of S_{DEST}. M(Q) is then the set of moves associated with the minimum-cost S_{DEST}.

We begin by expressing the query Q in a conjunctive form. In this form the condition expressed by every clause must be satisfied, and therefore, any clause which is one-variable (in V_2) represents local processing which is possible prior to any moves.

The first question that arises is this: are the one-variable clauses the only local operations that are

possible at this point? The answer is clearly no. It is possible to generate additional local operations by Transitivity. For example, in example 2.3 the clauses

(A.VQOH > C.YQty) AND (C.YQty > 1000)

imply

(A.VQOH > 1000)

which is a local operation (at site A) not represented in the original query. It follows, therefore, that we should close Q under transitivity. For example 2.3, the transitive closure has the form

Example 2.6

```
RETRIEVE (A.SS#)
WHERE (A.SS# = A.VS#) AND (A.SS# = C.YS#)
  AND (A.VS# = C.YS#) AND (A.VP# = B.PP#)
  AND (B.PP# = C.YP#) AND (A.VP# = C.YP#)
  AND (A.VQOH  > C.YQty) AND (C.YQty > 1000)
  AND (A.VQOH > 1000) AND (A.SCity = C.JCity)
  AND (B.PPname = 'Bolts') AND (C.JJ# = C.YJ#)
```

Given the transitive closure of Q (call it Q_1) we strip it of all local clauses (V_2 one-variable) and call the resulting query Q_2. Q_2 depicts the inter-site linkages in the original query Q and thus indicates the information that must be moved by the distribution strategy. In particular Q_2 specifies what columns of the V_2 relation at each site have to be retained in the distribution.

For Q_1 given by example 2.6, Q_2 is

Example 2.7

```
RETRIEVE (A.SS#)
WHERE (A.SS# = C.YS#) AND (A.VS# = C.YS#)
  AND (A.VP# = B.PP#) AND (B.PP# = C.YP#)
  AND (A.VP# = C.YP#) AND (A.VQOH > C.YQty)
  AND (A.SCity = C.JCity)
```

The columns to be retained are indicated below:

Example 2.8

Site	Columns to be Retained
A	(A.SS#, A.SCity, A.VS#, A.VQOH, A.VP#)
B	(B.PP#)
C	(C.YS#, C.YP#, C.YQty, C.JCity)

Now, we have assumed that only fragments of the original relations of the data model can be moved. In general, local operations at each site do not automatically yield such fragments. In example 2.8, the columns for sites A and C come from more than one original relation. Therefore, we need to perform the necessary projections to find the fragments which comprise M.

Example 2.9

Site	Fragments and Their Definition	
A	ASupplier (A.SS#, A.SCity) AAvailability (A.VS#, A.VP#, A.VQOH)	WHERE (A.SS# = A.VS#) AND (A.VQOH > 1000)
B	BParts (B.P#)	WHERE (B.PPname = 'Bolts')
C	CSupply (C.YS#, C.YP#, C.YJ#, C.YQty) CProject (C.JJ#, C.JCity)	WHERE (C.YQty > 1000) AND (C.JJ# = C.YJ#)

Note that the J# column has to be retained in CSupply and CProject in order to preserve the joining information C.JJ# = C.YJ#. The requirement that only fragments be moved implies that some joining operations would have to be repeated, hence those V_2 one-variable clauses have to be retained, e.g., the clauses (A.SS# = A.VS#) and (C.JJ# = C.YJ#) in example 2.9.

We can now readily determine M(Q), the minimum set of moves that can assemble all the data needed by Q at a single site with no further processing. Take each site in turn, and consider the cost of moving all the fragments not at that site.

Example 2.10

Final Destination	Fragments to be Moved
A	BParts, CSupply, CProject
B	ASupplier, AAvailability, CSupply, CProject
C	ASupplier, AAvailability, BParts

It is obvious that in example 2.10 B would never be chosen as the final destination. It is also likely that the fragment CSupply is very large since it involves the interrelationship among supplies, parts and projects. Therefore, for our example, the final destination is probably C and

$$M(Q) = \begin{Bmatrix} m_1 = \text{"move ASupplier from A to C"} \\ m_2 = \text{"move AAvailability from A to C"} \\ m_3 = \text{"move BParts from B to C"} \end{Bmatrix}$$

We can now summarize the algorithm for determining M(Q) as follows:

1. Put Q in conjunctive form.
2. Close Q under transitivity.

3. Perform local processing corresponding to V_2-one-variable clauses.
4. Identify the fragments at each site. And
5. Determine M(Q) by comparing costs.

2.3.5 Node Splitting in the Strategy Tree

The preceding section explained how M(Q), an initial distribution strategy is obtained. In this section we describe how M(Q) is successively refined.

The central step in the optimization procedure takes a proposed set of parallel moves, M, and replaces it by two successive sets of parallel moves, M_1 and M_2. As described in section 2.3.3, the procedure is first applied to the initial set of moves, M(Q), and then recursively to each move that replaces M(Q) and so forth. The procedure thus generates a binary tree whose nodes represent proposed sets of moves and whose leaves represent a sequence of moves that can be taken to process the query. The procedure terminates when it is not possible to replace any node in the tree by two sons whose total cost is less than the parent's cost.

The remainder of this section explores in greater detail the criteria for splitting a node, M, into two sons M_1 and M_2.

A node M which is not the root of the tree (i.e. a node which does not represent the initial set of moves, M(Q)) operates on a sub-query of the initial query Q. Let us call the sub-query which M is operating on, Q_1. Observe that Q_1 retains any clause which involves more than one fragment and not merely multi-variable clauses.

Example 2.11

$$M = \left\{ \begin{array}{l} m_1 = \text{"move ASupplier from A to C"} \\ m_2 = \text{"move AAvailability from A to C"} \\ m_3 = \text{"move BParts from B to C"} \end{array} \right\}$$

Q_1 is:

```
RETRIEVE (A.SS#)
WHERE (A.SS# = A.VS#) AND (A.SS# = C.YS#)
  AND (A.VS# = C.YS#) AND (A.VP# = B.PP#)
  AND (B.PP# = C.YP#) AND (A.VP# = C.YP#)
  AND (A.VQOH > C.YQty) AND (A.SCity = C.JCity)
  AND (C.JJ# = C.YJ#)
```

The optimization procedure dictates that M is split only if there is an immediate improvement. This implies that if M is split then the cost of at least one of the moves in M can be reduced. And the only way that can be done is

through a clause involving the fragment of that move. Hence, the first thing that we do is to identify the V_2 multivariable clauses which involve each of the fragments in M.

Example 2.12

move	fragment	clauses
m_1	ASupplier	(A.SS# = C.YS#) AND (A.SCity = C.JCity)
m_2	AAvailability	(A.VS# = C.YS#) AND (A.VP# = C.YP#) AND (A.VP# = B.PP#) AND (A.VQOH > C.YQty)
m_3	BParts	(A.VP# = B.PP#) AND (B.PP# = C.YP#)

The possible moves which might reduce each m_i are given by

move	potential reducing moves
m_1	m_{11} = CSupply [S#] from C to A, m_{12} = CProject [City] from C to A
m_2	m_{21} = CSupply [S#, P#, Qty] from C to A m_{22} = BParts [P#] from B to A
m_3	m_{31} = AAvailability [P#] from A to B m_{32} = CSupply [P#] from C to B

In order that a potential reducing move be included in M_1, its cost must be less than the reduction it effects.

Example 2.13

candidate	cost	reduction
m_{11}	CSupply [S#]	ASupplier reduced to (ASupplier(S#=S#)CSupply) [S#,City]
m_{12}	CProject [City]	ASupplier reduced to (ASupplier (City=City) CProject) [S#,City]
m_{21}	CSupply [S#, P#, Qty]	ASupplier reduced to (ASupplier(S#=S#)CSupply) [S#,City] and AAvailability reduced to (AAvailability (S#=S#,P#=P#,QOH>Qty) CSupply) [S#,P#,QOH]
m_{22}	BParts [P#]	AAvailability reduced to (AAvailability (P#=P#) BParts) [S#,P#,QOH]
m_{31}	AAvailability [P#]	BParts reduced to (BParts (P#=P#) AAvailability) [P#]
m_{32}	CSupply[P#]	BParts reduced to (BParts(P#=P#)CSupply) [P#]

It is obvious that m_{31} and m_{32} are bad moves, and m_{22} is an excellent move. It is likely that m_{21} is also a bad move since CSupply [S#, P#, Qty] is probably as large as AAvailability and much larger than the total reduction it effects. It is not clear as to whether m_{11} and m_{12} are worthwhile moves, and the precise answer would have to be provided by our estimate of the relative costs. Let's say that for our example m_{12} is worthwhile but m_{11} is not.

Now, we have determined certain moves which are not in M but which should be in M_1. Take these moves together with the moves of M not affected by them, and these comprise M_1. The remaining moves in M as modified by M_1 constitute M_2.

At this point we have to say something about whether multiple copies of a fragment are kept at different sites. For example, if we execute m_{22} = "move BParts from B to A", should BParts be retained at site B? If we do, m_3 in M is unaffected by m_{22}. If not, then m_3 becomes m_3' = "move BParts from A to C".

Example 2.14

For our example, M_1 is given by

$$M_1 = \left\{ \begin{array}{l} m_{22} = \text{"move BParts [P\#] from B to A"} \\ m_{12} = \text{"move CProject [City] from C to A"} \end{array} \right\}$$

The candidates to be included in M_2 are

m_1' = move (ASupplier(City=City)CProject) [S#,City] from A to C

m_2' = move (AAvailability(P#=P#)BParts) [S#,P#,QOH] from A to C

m_3' = move BParts from A to C

But m_3' can be considered to be redundant since it is a part of m_2'. Hence,

$$M_2 = \{m_1', m_2'\}$$

After the application of M_1 the residual query Q_2 is given by

```
RETRIEVE (A.SS#)
WHERE (A.SS# = A.VS#) AND (A.SS# = C.YS#)
  AND (A.VS# = C.YS#)AND (A.VP# = A.PP#)
  AND (A.PP# = C.YP#) AND (A.VP# = C.YP#)
  AND (A.VQOH > C.YQty) AND (A.SCity = A.JCity)
  AND (C.JJ# = C.YJ#)
```

where the clauses which have been changed from Q_1 are underlined.

If we do not make the move m_3' but subsume it in m_2', then it is tantamount to replacing (A.P# = C.YP#) by (A.VP# = C.YP#), which is already present in the query and hence can be eliminated.

The general procedure for node splitting can be summarized as follows:

a. The state at each node is represented by a V_2-query Q_1 and a proposed set of moves M on the fragments referenced by Q_1.

b. For each element m in an M we determine the set R_m of moves which would reduce the fragment referenced by m.

c. Take the union $\bigcup_m R_m$ and eliminate redundancy. Denote the resulting set by R.

d. Let M_1 be the set consisting of (i) any move in R which reduces more than it costs, and (ii) any move in M unaffected by the moves of category (i).

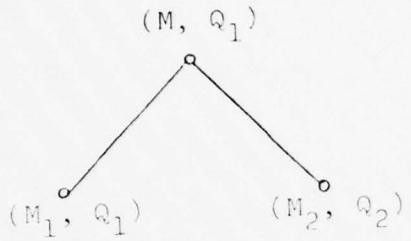
e. M is replaced by M_1 followed by

M_2 = {the remaining moves of M as modified by M_1}.

If $M=M_1$ the splitting stops.

f. Let Q_2 denote the query resulting from modifying Q_1 by M_1.

g. The node (M, Q_1) is replaced by $((M_1, Q_1), (M_2, Q_2))$ or



2.4 Reliability

2.4.1 Introduction

The problem that we address here is that of ensuring the reliable operation of SDD-1 in the presence of computer and communication failures. Any distributed, multi-site computer system must reflect in its fundamental design an appreciation of the fact that individual sites are certain to fail (and sometimes subsequently to recover) asynchronously with the rest of the system, and that loss of communication between sites is also bound to occur (whether as the result of accident or deliberate policy). Our overall goal is to develop mechanisms to detect such failures and respond to them, allowing the rest of the

system to continue operation, and then to reintegrate the system when the failure is repaired. In attempting to achieve this objective, we are guided by the following criteria:

a. simplicity . The principal feature of a reliability subsystem is that it must be reliable itself. To accomplish this, we have elected the emphasize simplicity in the design of our reliability mechanisms. That is, we take simple standardized steps on detecting a failure or a recovery, and eschew elaborate analyses. At times, this means that we forego the opportunity to detect and react to a special situation, but we believe the increased reliability of our procedures is worth the potential loss of optimal performance.

b. low overhead in normal conditions. It is not feasible for reliability mechanisms to be invoked only when an actual failure occurs; it is mandatory to take steps during normal operation to prepare for such eventualities. However, we have been guided by the principle that it is preferable to defer as much effort as possible to those occasions when failure processing must actually be invoked. This tradeoff means that the system

should perform better in normal circumstances, but will process failures and recoveries somewhat more slowly.

c. avoidance of global rollback. Often a failure will be detected while some transaction is in progress; it may then be necessary to abort that transaction and undo any of the actions it may have taken. Such rollback is a well-known and necessary reliability technique. However, we have been adamant that any such rollback should be confined to individual sites in the network; i.e., that rolling back a transaction at one site should not require the subsequent rolling back of a transaction run at another site. Such global rollback can readily propogate through a network and might even require the rolling back of long completed transactions. Our mechanisms are designed to avoid such a necessity.

d. paramaterizable degree of reliability. It is impossible to build a completely reliable distributed system (in a mathematical sense) out of unreliable components, unless one is willing to tolerate a crippling level of overhead. Our goal

has been to achieve very high degrees of reliability with an acceptable amount of machinery, and to structure the system so that this degree of reliability can be easily adjusted (at the expense of more mechanism). The amount of additional mechanism will depend on the reliability level desired and on the expected mean time between failures of the individual sites in the system.

e. operation in the Arpanet environment. Our system design has been influenced by the fact that SDD-1 will be expected to operate in an environment like the Arpanet. The implications of this assumption cut two ways. On the one hand, we can rely on certain failure detection facilities inherent in the Arpanet. However, several Arpanet mechanisms (especially with regard to inter-site communication) are not adequate for our purposes, and we have had to extend them, but not by making any modifications to the Arpanet software or structure itself.

2.4.2 Structure of the Problem

The most fundamental problem that we have had to address is how the problem of distriuted reliability should be structured for attack. We have divided the problems relating to reliability into four areas. The first of these is concerned with ensuring the reliable operation of the SDD-1 protocols in the presence of detected failures and recoveries of sites in the network. That is, assume that all aspects of the system operate reliably, except that sites may fail in a detectable way; the question then becomes, how should the remaining sites in the network react to learning of this failure, so that they can continue to operate in a correct way, and how can the failed site be reintegrated into the system when it recovers. At this level of abstraction, we assume that the units of failure are individual Global Data Managers and Local Data Managers. In the case of a failure of a GDM, there are two issues that need to be addressed: the effect of this failure on the transactions that the GDM is running at the time of its failure; and the effects of this failure on transactions that are running or will be run by other GDM's. (These latter effects can be caused

by the consistency interactions between the failed GDM and others.) In the case of LDM, the issues are the effect of the failure on running transactions, and the effect of the failure on the protocols that GDM's will have to run for transactions that would be otherwise reading from the failed LDM.

The second area of reliability is concerned with providing the level of abstraction that the first set of problems was assuming; namely, a correctly operating network, whose only failures are detected failures of LDM's and GDM's. We have termed the set of support mechanisms that seek to achieve this capability a reliable network. The major requirements of the reliable network are the following:

a. Every message sent from one site to another will eventually be received at the destination site. This requirement is clearly demanded if the protocols are to operate correctly. Problems arise in achieving it in the context of site failures. For example, suppose A attempts to send a message to B when B is in a failed state ("down"), and A itself fails before B recovers. How does B receive the message?

b. The order transmission of two messages from one site to another is identical to the order of their receipt. This is an explicit assumption of the protocols, and is not automatically provided in a network where there are multiple routes between two sites.

c. If any updates associated with a transaction run by a GDM are sent to LDM's, then all the update messages are. This demand must be met if the system as a whole is to see a uniform and consistent view of the transaction; otherwise, some sites will believe that the transaction has run, and others will not. The problem is to meet this demand when it is possible for the GDM to fail after having sent out some, but not all, of the update messages associated with a transaction.

d. The failure of a site becomes known to all other sites that may be affected by the failure by the time that they need to know of it, and after they have received any messages the site may have sent out before it failed. This is one of the explicit assumptions made at the higher level of abstraction discussed earlier. The two problems that must be

addressed here are first, having each appropriate site detect the failure uniformly, reliably, and promptly; and second, coordinating this failure detection with other activities, especially the receipt of messages.

e. Upon its recovery, a site should institute recovery procedures, and all other sites should learn of it promptly. The problem issues are ensuring that a site knows that it is recovering from a failure, and accounting for brief failures, where a site may recover before others have learned that it has failed.

The third area of concern is for the low-level mechanisms to enable individual sites to detect and service failures. Issues here include preserving relevant state information in the case of a failure, implementing appropriate local rollback, and detecting other site failures in a timely fashion. Most of the problems in this domain are analogous to recovery and reliability concerns in conventional data management systems; the rest are principally concerned with the interface of the distributed data base system with the operating system that supports it.

The final problem area concerns the issue of network partitioning. This means that, because of communication failures, it becomes impossible for some operating sites in the network to communicate with others; i.e., the network becomes several sub-networks, with no communication between them. The difficulty that this presents is that, without communication, there is no way for independently operating computers to coordinate thier activities. Therefore, where the full network is reconnected, various copies of the database may be arbitrarily inconsistent; the SDD-1 protocols that assure consistency depend on complete communication between operating sites.

The only purely "technical" solutions to this problem are unacceptable from a practical point of view; they include such approaches as forbiding some sites from running transactions when the network is partitioned, to unrolling all the transactions run by some sites at the time the network is reconnected. Consequently, our approach is based on explicit human intervention at the time the partition is discovered and at the time it is restored. Our research has uncovered a variety of scenarios that a Data Base Administrator might wish to follow, depending on the nature of the data base and the transactions with it. That is, the appropriate strategy depends on the semantics

of the application, and so does not admit of automation of this time. Therefore, the major issues to be addressed in this area include determining the range of possible responses to a partition situation, developing techniques for the detection of a partition and of its restoration and providing the DBA with tools to enforce his policy decisions. Examples of the strategies that the DBA might wish to follow include designating one copy of a fragment as the correct one, from which all others are to be copied, to running some complex procedure to resolve the inconsistencies among different versions.

Our principal emphasis to date, has been on the first two problem areas discussed above: robust operation of the protocols, and design of reliable network mechanisms. In the next section, we present some of the techniques that we have developed in these areas.

2.4.3 Reliability mechanisms

In this section, we summarize some of the particular mechanisms that have been developed to address specific reliability issues in SDD-1.

2.4.3.1 Reformulation

When an LDM fails, any transactions in progress that attempt to read data from a fragment stored at that site will be aborted - i.e., any partial results will be discarded and any invoked activities will be suspended. However, in the future, such transactions will have to read that data from another LDM. The process of deciding which LDM will be used to replace one that has failed is called reformulation, and it raises the possibility of major changes in the protocols that various transactions will be run under, because of changes in the graph topology. The most disturbing possibilities are that the failure of a LDM might cause such extensive changes in the graph structures that even transactions that had no interaction with the failed LDM might have to change their

protocol, or that the entire graph needs recomputation in order to determine the appropriate protocol changes. Our investigations have demonstrated that neither of these eventualities will transpire. First of all, the only transactions whose protocols will change are those that attempt to read from the failed LDM. This means that not all GDM's need to be informed of the failure of an LDM, only those that normally utilize it; this minimizes the amount of failure detection and communication apparatus that is needed. Second, the changes in protocol of a transaction can be totally determined from information available at the GDM at which the transaction runs. These two facts allow LDM failure to be handled in a distributed fashion, enhancing the simplicity and reliability of the system.

2.4.3.2 Synchronization with a Failed GDM

Several of the protocols described earlier operate by attaching conditions on the read messages sent by the GDM to an LDM. This condition specifies the LDM is to wait until it has received updates from some other specified GDM's at a specified time, and then return the requested data to the requesting GDM. A difficulty arises if some of the GDM's for which the LDM is waiting have failed. It is infeasible for the LDM to wait until they have recovered before returning the requested data; yet upon recovery, one of the failed GDM's may issue updates timestamped prior to the time through which the LDM was instructed to wait. We resolve this issue by having the LDM , upon detection of the failure of one of the GDM's for which it is waiting, send that GDM a message containing the time through which the LDM was instructed to wait; the LDM can then proceed under the assumption that it has received all relevant updates from that GDM. Upon recovery, the GDM retrieves all these messages, and sets its own time-clock to be marginally later than the latest time on any of them. In this way, the GDM is guaranteed not to violate any of the assumptions made by

an LDM during the time it was down. The reliable message delivery mechanism, described below, assures that the GDM will receive these messages even if the issuing LDM's fail before the GDM recovers.

2.4.3.3 Reliable Message Delivery

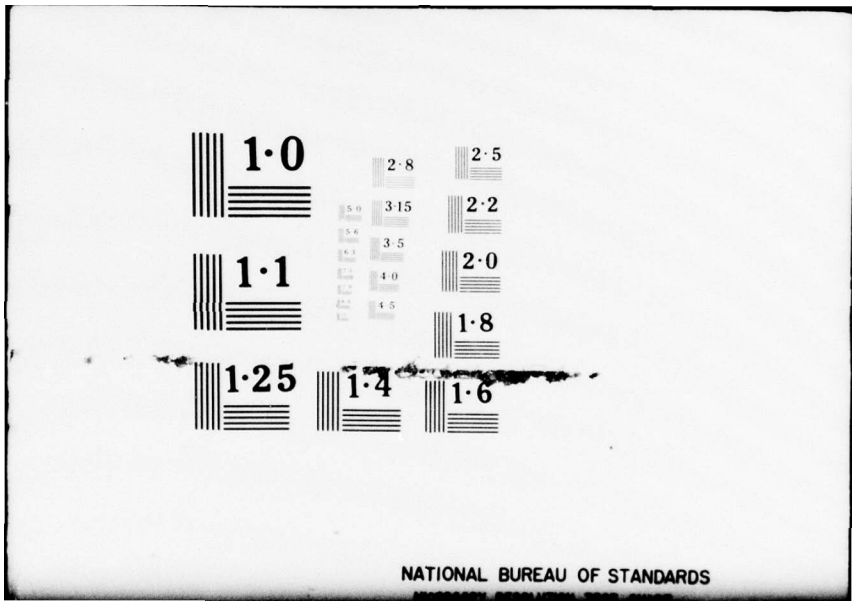
As we have earlier observed, it is mandatory that any messages sent from one SDD-1 site to another eventually arrive at the destination, even if the source and destination may fail at arbitrary times. Our approach to this problem is based on the notion of message spoolers. If site A attempts to send a message to site B, but is unable to do so because B is currently down, then A will establish copies of this message with a number of message spoolers, which reside at a variety of sites. (Usually, A will maintain one of these spoolers itself.)

When B recovers from its failure, it picks up the messages sent it while it was down from one of the spoolers. The need for a multiplicity of spoolers derives from the possibility that a spooler itself might fail before B recovers. This is addressed by having the spoolers monitor each other on a periodic basis; when the failure of a spooler is detected, the remaining ones invoke a new

AD-A044 441 COMPUTER CORP OF AMERICA CAMBRIDGE MASS F/G 9/4
A DISTRIBUTED DATABASE MANAGEMENT SYSTEM FOR COMMAND AND CONTRO--ETC(U)
JUN 77 N00039-77-C-0074
UNCLASSIFIED CCA-76-06 NL

2 OF 2
ADA
044441

END
DATE
FILMED
10-77
DDC


1·0
1·1
1·25
1·4
1·6
1·8
2·0
2·2
2·5
2·8
3·15
3·5
4·0
4·5
5·0
5·6
NATIONAL BUREAU OF STANDARDS

one and transmit to it all the currently spooled messages. In this way, the quota of active spoolers can be maintained. The number of spoolers can be adjusted to achieve a desired level of reliability, given an expected frequency of site failure.

2.4.3.4 Broadcast of Updates

It is essential that SDD-1 preserve the atomicity of transactions in the presence of site failures. The major problem in this regard is that a GDM might fail in the process of sending out the updates to the appropriate LDM's, after the completion of some transaction; if some, but not all, of these updates are issued, then different copies of the same data may be left in inconsistent states. We address this issue by causing update messages to be transmitted in two stages. The first stage is the actual update message itself, which contains the new values to be installed as a result of the transaction. After all of these messages have been sent out, the GDM sends out an installation notice to each of the LDM's; it is only upon receipt of this second message that the LDM will actually perform the updates specified by the first message. If an LDM receives an update message and does

not receive the associated installation notice within a reasonable period of time, it will enquire of the other LDM's if they have received a notice. If it receives an affirmative response, it too installs the update; otherwise, it discards it.

Thus, if the GDM failure occurs before all the updates have been sent out, then no installation notices will have been sent, and so these updates that were sent will eventually be discarded. If the GDM failure occurs after some installation notices have been sent, then all the updates will have been sent; and those LDM's not having received installation notices will be able to communicate with LDM's that have and know that the update is to be installed. To assure this communication ability, we further require that the first few LDM's to receive installation notices duplicate them and send them on to other LDM's. In this way, it will be possible for any LDM to communicate with some LDM that has received an installation notice, even in the presence of LDM failures.

3. Activities at NOSC

In addition to the design work described in the previous section, three software systems were installed at NOSC. These systems were: the Datacomputer, the Terminal Access Program (TAP) and the initial Alerting capability. CCA also provided extensive assistance to NOSC in using the Datacomputer, improving Datacomputer performance and installing the Bluefile at NOSC.

3.1 Installation of Initial Datacomputer

The Datacomputer at CCA was converted to run under standard TENEX and was installed at NOSC in January. In addition to installing and testing the Datacomputer, CCA assisted NOSC by providing a set of distance functions and by moving the entire Bluefile database from the CCA Datacomputer to the NOSC Datacomputer. These distance functions enable the Datalanguage program to conveniently compute the distance between two ships in the Bluefile. This facility was considered vital to the Datacomputer application at NOSC. CCA also assisted NOSC by helping

them improve their Datacomputer usage and making Datacomputer changes that were considered desirable by CCA and NOSC.

3.2 TAP

The Terminal Access Program (TAP) was delivered to NOSC in late March. TAP is a general purpose program which allows on-line users to access any part of the database. TAP was developed for handling needs outside the scope of pre-planned applications, for gaining familiarity with both the system and the data it contains, and to serve as a demonstration and test vehicle for the database.

TAP deals with a relational database. Each relation is a single Datacomputer file, directly under the user's LOGIN node. The data associated with a relation consists of sets of attribute-value pairs, or tuples. Attributes are equivalent in meaning to Datalanguage container names of type STRING or INTEGER. No relation will contain duplicate copies of the same tuple.

Each relation has a set of one or more attributes defined as its key; these attributes are made sub-containers of a STRUCTURE named KEY. Keys are unique within a relation.

All attributes are fixed length strings or integers. The following sections describe TAP functions and commands.

3.2.1 Initialization

TAP creates a script file for the session, establishes a connection to the Datacomputer, prompts the user for a name and password, and generates an appropriate LOGIN command.

3.2.2 ADD

TAP prompts the user for a relation and a source of tuples. Note that a selection is not valid here. Tuples will be accepted from the source, checked for uniqueness in the relation, and appended to it or objected to as appropriate. TAP assumes that the user will not enter duplicate tuples in a single ADD command; that is, the uniqueness test will only be applied against the relation as it was before the ADD began.

If the user specifies the terminal as the source of tuples, each attribute will be prompted. Alternatively, a TENEX disk file may be specified, whose format should

correspond exactly to the file stored in the Datacomputer. Tuples are transferred from the file without further interaction.

3.2.3 CHANGE

CHANGE changes attribute values in existing tuples in a relation. It prompts for a set of tuples to which the change is to apply. If the user enters a relation, or a selection with a null qualification, it prompts for a selection criterion. Then it begins a dialogue to specify which attributes in selected tuples are to be changed, and what their new values are to be. When all attributes have been specified, the update is carried out: for each selected tuple, the attributes which had new values specified receive those values, destroying their old values. Attributes which did not have values specified are left untouched. Key attributes and attributes whose values are Virtual Expressions (indicated by "VE=...") cannot be changed.

3.2.4 COUNT

TAP prompts for a relation/selection (R/S); it will return to the user the number of tuples in that R/S.

3.2.5 CREATE

TAP requests a name for the new relation, and an existing relation to use as a model. If a model is given, its description will be copied; if not, the user is given an empty shell of a relation. This description may then be modified by adjusting file-level parameters and adding and deleting attributes, until the user is satisfied. TAP then creates the new relation in the Datacomputer.

3.2.6 DESTROY

TAP prompts for a relation or selection name. If a relation name is typed in, that relation will be deleted from the Datacomputer. If a selection name is entered, the selection will be flushed from TAP's selection list.

3.2.7 DISPLAY

TAP prompts the user for a relation/selection (R/S). The values of the attributes of each tuple in that R/S will be displayed on the user's terminal. Since all records are fixed-length, strings which are shorter than that length are padded with blanks; integers, with zeros. Each tuple is printed on two lines; the values of the key fields of the tuple are printed on the first line, and values of the non-key fields are printed on the second (indented) line. Each value is followed by a comma.

3.2.8 HELP

HELP describes any of the TAP action commands, such as ADD, CHANGE, SELECT, etc. To get the help text for a particular command, just type enough of the command to identify it uniquely. E.g., typing "A" will get the description of the ADD command, but "CH", "CO", or "CR" must be typed to identify CHANGE, COUNT or CREATE, respectively.

Help text is paged; that is, if the text is longer than one screenful, HELP will print one screeen's worth of data and then wait for a response (<CR>) from the user before proceeding.

3.2.9 JOIN

TAP prompts for a first and second relation/selection (R/S), a destination name, and for a set of matching criteria -- pairs of attributes from the first and second R/S's with equal values. From these it will construct a new relation. If the attributes of a matching pair do not have the same name, the name of the first will be used in the destination. If non-matching attributes in the two sources have the same name, TAP prompts for a new name for the second relation's attribute. The key of the new relation will be the concatentation of the keys of the two sources; except, if all key attributes of the second source are matched, the key will be the same as the first source's. TAP then fills the new relation with the JOIN of the two source R/S's: for each member of source-1, it will find all members of source-2 which match on the specified attributes. For each of these, it will generate a tuple in the destination, with values taken from the two sources.

3.2.10 LIST

LIST permits the TAP user to view various components of the TAP's data base, including: relation and selection names, parameters, keys or attributes of a given relation, and qualifiers of exisiting selections. After typing LIST, TAP replies with a "**" prompt. The user may type one or more letters of the following subcommands (followed by <CR>, <ESC> or space):

Attributes (of relation)

Key (attributes of relation)

Parameters (of relation)

Relation (name list)

Selection (name list)

3.2.11 MOVE

MOVE copies tuples from one relation into another. It prompts for a source relation or selection (if a selection, only selected tuples will be copied); a destination relation, and a mode. If the mode is APPEND, tuples are added to those already in the destination, after being checked to insure they do not duplicate existing destination tuples. If the mode is OVERWRITE, all existing tuples in the destination are deleted, and then all (selected) tuples in the source are nondestructively transferred. Every source-key attribute must correspond to a like-named destination-key attribute.

MOVE append mode requires that the KEY of source and destination consist of the same attributes. Append mode requires that the source and destination keys must be identical with respect to number of attributes and attribute names.

Duplicate tuples will be rejected by the Datacomputer. If any duplicates are found, TAP will print an advisory message one time only, indicating that at least one duplicate tuple was found and rejected. The actual

rejected tuples can be found in the user's script file after the TAP session.

3.2.12 PROJECT

TAP prompts the user for a source relation or selection, a destination name, and a set of attributes to be eliminated. It will create the new relation and fill it from the source, eliminating any duplicate tuples.

3.2.13 QUIT

After requesting confirmation from the user, TAP will terminate its session with the Datacomputer and return to TENEX.

3.2.14 REMOVE

REMOVE deletes tuples from a relation. The user specifies a relation and a qualification. After receiving confirmation, TAP will delete from that relation all tuples satisfying the given qualification. A null qualification means "all tuples."

3.2.15 SELECT

TAP prompts for a relation, a qualification, and a new name; from these it will construct a reference to a subset of a relation. No data will actually be manipulated, but the new name (hereafter called a SELECTION) may be used in place of a relation name in many of TAP's functions. The qualification must be a valid Datalanguage WITH clause, specifying a boolean combination of relational terms of the form

relation-attribute relatonal operator constant-value.

"Relation" may be a name of an existing relation. In this case, "new name" will refer to the subset of that relation

which satisfies "qualification". "Relation" may also be the result (i.e. "new name") of a prior selection. That is, selections may be chained, yielding a set-intersection. In this case, the new name will specify the conjunction of the current qualification with the qualification of the old selection, all applied to the original relation.

3.3 Initial Alerting Capability

In late March a new version of the Datacomputer was delivered to NOSC. This version included the initial alerting capability.

Two new Datalanguage commands were implemented in support of the alerting capability:

```
WATCH <pathname>;
UNWATCH <pathname>;
```

The user has the capability of 'watching' up to 10 files at any one time. Four messages may be received by a user who is 'watching' one or more files. If another user is modifying <pathname> at the time the WATCH command is issued, the following message is produced.

!A290 UPDATE IN PROGRESS FOR FILE = <pathname>

Upon starting execution of a request which modifies a file, all watchers of that file receive the following message.

!A290 UPDATE STARTED FOR FILE = <pathname>

Upon successful completion of such a request, all watchers receive the following message.

!A291 UPDATE COMPLETED FOR FILE = <pathname>

If such a request terminates unsuccessfully, all watchers will receive the following message.

!A292 FLUSHING UPDATE OF FILE = <pathname>

A user will never receive one of the above messages as a result of his own modification. The update will not be visible until the update completed (!A291) message has been received.

The LIST command was expanded in support of the alerting capability. The 'STAT=' field of the %STATUS option has been augmented to indicate if the user has issued a WATCH command for a particular file. It does not indicate that another user is watching the file. The output for a 'watched' file is:

STAT = ONLINE-WATCH

or

STAT = OFFLINE-WATCH

The LIST command accepts %WATCH in place of <pathname> and performs the specified option for all files for which the WATCH command was issued.

LIST %WATCH %SOURCE;

outputs the source for all files which are being watched by this user.

3.4 TOPS20 Datacomputer

In early June a new version of the Datacomputer was installed by CCA personnel at NOSC. This Datacomputer is capable of running under both TOPS20 and standard TENEX.

Other major changes in this Datacomputer include:

- increased use of inversions for retrievals based on non-constant values
- significant reduction in the time required to load files

- significant reduction in the data storage overhead for chaptered files.

3.5 Consultation

In addition to general phone consultation with NOSC personnel, the CCA staff performed several specific tasks:

- CCA has developed TENEX subroutines which handle common cases of application - Datacomputer interaction. During this reporting period, this set of subroutines was incorporated into a Fortran interface package (called DCPKG) usable from NOSC application programs.

- In April, CCA personnel were on site at NOSC to aid in the initial loading of the FC database.

- In June, CCA personel conducted extensive experiments at NOSC to determine process performance and competition for machine resources.

3.6 Documentation

The following list of documentation was delivered to NOSC:

- Datacomputer User Manual and Guides
- TAP User Guide
- Datacomputer Overview
- Datacomputer's Operator's Manual
- Datacomputer Programming Environment
- Directory System Overview
- Buffering Monitor
- RH Module Overview
- Intermediate Language Specification
- Compiler Service Routines
- RH Error Recovery
- How to Read an RH Dump
- RH Error Recovery

4. Meetings and Publications

During the reporting period members of the SDD-1 project staff made presentations describing the system design at:

Defense Mapping Agency

Defense Communications Engineering Center

Rome Air Development Center

Database Technology for DOD Managers (seminar sponsored by Command and Control Technical Center at Pentagon)

National Bureau of Standards

Harvard University

Northwestern University

Massachusetts Institute of Technology

Princeton

University of Toronto

University of California, Berkeley

In addition, the following technical reports describing this design were produced:

CCA-77-01 A Study of Updating in a Redundant Distributed Database Environment

J. Rothnie, N. Goodman

February 15, 1977

CCA-77-02 The Redundant Update Methodology of SDD-1: A System for Distributed Databases (The Fully Redundant Case)

J. Rothnie, N. Goodman, P. Bernstein

June 5, 1977 (DRAFT)

CCA-77-03 Retrieving Dispersed Data from SDD-1: A System for Distributed Databases

G. Wong

May, 1977

(Also published in Proceedings of the 1977 Berkeley Workshop on Distributed Data Management and Computer Networks)

CCA-77-04 An Overview of the Preliminary Design of SDD-1: A System for Distributed Databases

J. Rothnie, N. Goodman

May, 1977

(Also published in Proceedings of the 1977 Berkeley Workshop on Distributed Data Management and Computer Networks)

CCA-77-05 Analysis of Serializability in SDD-1: A System for Distributed Databases (The Fully Redundant Case)

References

[ALSBERG et al]

Alsberg, P.A.; Belford, G.G.; Bunch, S.R.; Day, J.D.; Grapa,E.; Healy, D.C.; McCauley, E.J.; and Willcox, D.A. Synchronization and Deadlock, CAC Document Number 185, CCTC-WAD Document Number 6503, Center for Advanced Computation, University of Illinois at Urbana-Champagne, Urbana Illinois, March 1, 1976. (see also, Research in Network Data Management and Resource Sharing: Final Research Report, CAC Document Number 210, CCTC-WAD Document Number 6508, Center for Advanced Computation, University of Illinois at Urbana-Champaign, Urbana Illinois, September 30, 1976.)

[ALSBERG and DAY]

Alsberg, P.A.; and Day, J.D. "A Principle for Resilient Sharing of Distributed Resources", Report from the Center for Advanced Computation, University of Illinois at Urbana-Champaign, Urbana Illinois, 1976. (Also accepted for proceedings of the Second International Conference on Software Engineering.)

[ASTRAHAN et al]

Astrahan, M. M.; et al. "System R: Relational Approach to Database Management", ACM Transactions on Database Systems, Vol. 1, No. 2, June 1976, pp. 97-137.

[BERNSTEIN et al a]

Bernstein, P.A.; Rothnie, J.B.; Goodman, N.; and Papadimitriou, C.A. "Analysis of Serializability in SDD-1: A System for Distributed Databases (The Fully Redundant Case)", First International Conference on Computer Software and Applications (COMPSAC 77), IEEE Computer Society, Chicago Illinois, November 1977. (Also available from Computer Corporation of America, 575 Technology Square, Cambridge, Massachusetts 02139 as Technical Report No. CCA-77-05).

[BERNSTEIN et al b]

P. A. Bernstein; J. B. Rothnie; D. Shipman; N. Goodman. "The SDD-1 Redundant Update Algorithm" Technical Report CCA-77-06, Computer Corporation of America, 575 Technology Square, Cambridge, Massachusetts 02139.

[CHAMBERLIN et al a] Chamberlin, D.D.; Boyce, R. F.; Traiger, I.L. "A Deadlock-free Scheme for Resource Locking in a Database Environment", Information Processing 74, Proceedings AFIPS Conference, North Holland Publishing Company, Amsterdam The Netherlands, 1974.

[CHAMBERLIN et al b]

Chamberlin, D.D.; Gray, J.N.; and Traiger, I.L. "Views, Authorization, and Locking in a Relational Database System", Proceedings AFIPS National Computer Conference, AFIPS Press, Vol. 44, 1975.

[CHU]

Chu, W.W. "Optical File Allocation in a Computer Network", in Computer Communication Networks, Kuo, F.F. editor, Prentice-Hall Computer Applications in Electrical Engineering Series, Prentice-Hall Inc., Englewood Cliffs NJ, 1973.

[ELLIS]

Ellis, C.A. "A Robust Algorithm for Updating Duplicate Databases", 1977 Berkeley Workshop on Distributed Data Management and Computer Networks, Lawrence Berkeley Laboratory, University of California, Berkeley California, May 1977.

[ESWARAN et al]

Eswaran, K.P.; Gray, J.N.; Lorie, R.A.; Traiger, I.L. "The Notions of Consistency and Predicate Locks in a Database System", Communications of the ACM, Vol. 19, No. 11, November 1976.

[FRY and SIBLEY]

Fry, J. P.; and Sibley, E. H. "Evolution of Database Management Systems", Computing Surveys, Vol. 8, No. 1, March 1976, pp 7-42.

[GRAY et al]

Gray, J.N.; Lorie, R.A.; Putzolu, G.R.; Traiger, I.L. "Granularity of Locks and Degrees of Consistency in a Shared Database", Report from IBM Research Laboratory, San Jose California, 1975.

[GREIF]

Greif,I. Semantics of Coordinating Parallel Processes, Technical Report No. TR-154, Laboratory for Computer Science, M.I.T., Cambridge Massachusetts, September 1975.

[HAMMER and McLEOD]

Hammer, M.M.; and McLeod, D.J. "Semantic Integrity in a Relational Database System", Proceedings of the International Conference on Very Large Data Bases, September 1975.

[HEWITT]

Hewitt, C.E. "Protection and Synchronization in Actor Systems", Artificial Intelligence Laboratory Working Paper No. 83, Massachusetts Institute of Technology, November 1974.

[LAMPORT]

Lamport, L. "Time, Clocks and Ordering of Events in a Distributed System" Computer Associates Report #CA-7603-2911, March 1976. Also submitted to CACM.

[MARILL and STERN]

Marill, T.; and Stern, D. H. "The Datacomputer: A Network Data Utility", Preceedings AFIPS National Computer Conference, AFIPS Press, Vol. 44, 1975.

[METCALFE]

Metcalfe, R.M. Packet Communication, Technical Report No. TR-114, Laboratory for Computer Science, M.I.T., Cambridge Massachusetts, December 1973.

[ROTHNIE et al]

Rothnie, J.B.; Goodman, N.; and Bernstein, P.A. "The Redundant Update Algorithm of SDD-1: A System for Distributed Databases (The Fully Redundant Case)", First International Conference on Computer Software and Applications (COMPSAC 77), IEEE Computer Society, Chicago Illinois, November 1977. (Also available from Computer Corporation of America, 575 Technology Square, Cambridge, Massachusetts 02139, as Technical Report No. CCA-77-02).

[ROSENTHAL]

Rosenthal, R. "A Review of Network Access Techniques with a Case Study: The Networks Access Machine", NBS Technical Note 917, July 1976.

[ROTHNIE and GOODMAN a]

Rothnie, J.B.; and Goodman, N. "A Study of Updating In a Redundant Distributed Database Environment", Technical Report No. CCA-77-01, Computer Corporation of America, 575 Technology Square, Cambridge Massachusetts 02139, February 15, 1977.

[ROTHNIE and GOODMAN b]

Rothnie, J.B.; and Goodman, N. "An Overview of the Preliminary Design of SDD-1: A System for Distributed Databases", 1977 Berkeley Workshop on Distributed Data Management and Computer Networks, Lawrence Berkeley Laboratory, University of California, Berkeley California, May 1977. Also available from Computer Corporation of America, 575 Technology Square, Cambridge Massachusetts 02139, as Technical Report No. CCA-77-04).

[SCHANTZ and MILLSTEIN]

Schantz, R. E. and Millstein,R. E. "The FOREMAN: Providing the Program Execution Environment for the National Software Works", BBN Report No. 3266, March 1976.

[SIBLEY]

Sibley, E. H. "The Development of Database Technology", Computer Surveys, Vol. 8, No. 1, March 1976, pp1-5.

[STONEBRAKER and NEUHOLD]

Stonebraker, M.; and Neuhold, E. "A Distributed Database Version of INGRES", 1977 Berkeley Workshop on Distributed Data Management and Computer Networks, Lawrence Berkeley Laboratory, University of California, Berkeley California, May 1977.

[THOMAS a]

Thomas, R. H. "A Resource Sharing Executive for the Arpanet", Proceedings AFIPS National Computer Conference, AFIPS Press, Vol. 42, 1973, pp. 155-163

[THOMAS b]

Thomas, R.H. "A Solution to the Update Problem for Multiple Copy Databases Which Uses Distributed Control", BBN Report No. 3340, Bolt Beranek and Newman Inc., Cambridge Massachusetts, July 1975.

[WONG]

Wong, E. "Retrieving Dispersed Data from SDD-1: A System for Distributed Databases", 1977 Berkeley Workshop on Distributed Data Management and Computer Networks, Lawrence Berkeley Laboratory, University of California,

Berkeley California, May 1977. (Also available from Computer Corporation of America, 575 Technology Square, Cambridge Massachusetts 02139, as Technical Report No. CCA-77-03).