THE DESIGN
OF A
DISTRIBUTED DATABASE
AND
A REPLICATED DATA MANAGEMENT ALGORITHM

by

Steven J. Van Buren

B. S., Michigan Technological University, 1972

A REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1988

Approved by:

[Signature]
Major Professor
# CONTENTS

## CHAPTER 1 INTRODUCTION

1.1 The Financial Control System .................................. 1
1.2 System Architecture ........................................... 2
1.3 Autonomy ..................................................... 4
1.4 Related Work .................................................. 4
1.5 Organization of Paper ......................................... 7

## CHAPTER 2 DESIGN OF THE GLOBAL SCHEMA

2.1 Introduction ................................................... 9
2.2 Analysis of Requirements ..................................... 10
  2.2.1 The Problem ............................................... 10
  2.2.2 Output Requirements ..................................... 11
    2.2.2.1 Timeliness ......................................... 11
    2.2.2.2 Accuracy ........................................... 11
    2.2.2.3 Presentation of Output ........................... 12
  2.2.3 Other Requirements ....................................... 12
    2.2.3.1 Data Entry ......................................... 13
    2.2.3.2 Constraints ......................................... 13
    2.2.3.3 Application Maintenance and Development .... 14
    2.2.3.4 Documentation .................................... 15
    2.2.3.5 User Support ....................................... 15
  2.3 The Global Schema .......................................... 15
    2.3.1 Identifying Data Elements ............................ 16
    2.3.2 Normalization .......................................... 17
    2.3.3 Further Normalization .................................. 19
      2.3.3.1 Employee Table .................................. 20
      2.3.3.2 Mord_nos Table ................................ 21
      2.3.3.3 Dept_budg Table ................................ 21
      2.3.3.4 Rates Table ...................................... 22
      2.3.3.5 Spafactor Table ................................ 22
      2.3.3.6 Timsheet Table ................................ 23
      2.3.3.7 Tim_detail Table ................................ 24
      2.3.3.8 Payperiod Table ................................ 24
      2.3.3.9 Cont_time and Cont_detail Tables ............ 24
      2.3.3.10 Voucher Table .................................. 25
      2.3.3.11 Purch_ord Table ................................ 26
      2.3.3.12 Pettycash Table ................................ 26
      2.3.3.13 GN300 Table .................................... 27
  2.4 Evaluating the Global Schema ............................... 27
  2.5 Summary Table ............................................... 29

## CHAPTER 3 DISTRIBUTION DESIGN

3.1 Introduction .................................................. 30
3.2 Fragmentation ................................................ 30
  3.2.1 Horizontal Fragmentation ................................ 30
  3.2.2 Vertical Fragmentation ................................. 31
3.3 Allocation .................................................... 32
  3.3.1 Full Replication ....................................... 32
LIST OF FIGURES

Figure 1. Ocean Systems Organization .......................................................... 3
Figure 2. Replicated Table Update Transaction ............................................... 44
Figure 3. Algorithm Structure ......................................................................... 45
Figure 4. Financial Control System Structure .................................................. 46
Figure 5. Global Schema Employee & Mord_nos ............................................. 70
Figure 6. Global Schema Dept_budg, Rates, & Spafactor ................................. 71
Figure 7. Global Schema Timesheet, Tim_detail, & Payperiod .......................... 72
Figure 8. Global Schema Cont_time & Cont_detail ......................................... 73
Figure 9. Global Schema Voucher & Purch_ord ............................................... 74
Figure 10. Global Schema Pettycash & GN300 ................................................. 75
Figure 11. Global Schema Summary Table ..................................................... 76
LIST OF TABLES

TABLE 1. Incurrence Categories ............................................. 12
TABLE 2. Distribution Design .............................................. 34
TABLE 3. Supporting Relations for Algorithm ............................. 39
ACKNOWLEDGEMENTS

I wish to express my appreciation to my Department Chief, Mr. G. M. Smith Jr. and my Assistant Manager, Mr. G. B. Manning for allowing me the time to work on the design of the Financial Control System. Without their cooperation, the completion of this project would not have been possible.

I also thank my wife Kim for having faith in me and supporting me during the development of this report. She managed a full-time job along with caring for our daughter Rachel; all so Daddy could work on his Master's report.
Databases have proven themselves invaluable in the day to day operation of corporate America. As databases and their importance has grown, so has database technology. Database design techniques have made important advances. The software systems which manage and maintain the data (database management systems) have been greatly improved. Hardware has been specifically designed to support database applications (database machines). Most of this effort has been directed toward centralized databases - those which reside on a single, usually large, computer.

The advent of improved computer to computer communications, and the work on distributed systems has been reflected in database technology as distributed databases. A distributed database is basically a number of centralized databases connected by communications links, therefore the advances made in centralized databases are not lost, they are built upon.

The apparent trend in corporate America is merger and takeover. This situation creates an atmosphere which is ideally suited to the concepts of distributed databases. Dispersed (geographically or functionally) divisions within a corporation should have more local control over their own data yet still cooperate, on a higher level, for the good of the overall corporation. The distributed database concept can meet these needs.
1.1 The Financial Control System

The Ocean Systems Organization of AT&T Technologies' Federal Systems Division manages the operations of various Federal contracts. The purpose of this report is to describe the design of a distributed database which will be used to manage and track the incurrences on these contracts. Some basic algorithms for insuring the consistency and integrity of replicated data will also be designed. The distributed database will be built on top of the INFORMIX-SQL central database management system (DBMS). The project is titled the Financial Control System.

The Ocean Systems Organization is hierarchically structured along functional lines. (See Figure 1) The Financial Control System will be designed to match the organization's hierarchical structure. There will be a local database established at each of the assistant manager nodes of Figure 1. These assistant manager databases will reside on computers termed the local nodes. Each local node will have a number of departments under it. The departments will be the source of all the raw financial data. Figure 1 is a good representation of raw input data flow from department to local node database. From each local node, summary data will be generated and will flow up to the central node where it will be aggregated.

1.2 System Architecture

A distributed database is a collection of data that belong logically to the same system, but are spread over multiple computers connected by a network [CERI84]. Distributed database management systems (DDBMS) are classified as homogeneous or heterogeneous. Heterogeneous DDBMSs are defined as having at least two different database management systems installed at two different nodes in the network. Conversely, a homogeneous DDBMS has the same local DBMS installed at
Figure 1. Ocean Systems Organization
each node. The Financial Control System will be a homogeneous DDBMS since the DBMS installed at each node will be INFORMIX-SQL™. INFORMIX-SQL is a relational DBMS which runs under UNIX™ as well as other operating systems. Each node in the Financial Control System will be a UNIX based computer tied into an existing local area network which provides point-to-point communications.

1.3 Autonomy

One of the most important issues of distributed database administration is the degree of local autonomy given to each node. There are two extreme solutions, the absence of any local autonomy and complete local autonomy [CERI84]. The goal of the Financial Control System is to provide as much local autonomy as possible. It will not be possible to design the system with complete local autonomy since some of the data required by the local nodes is not available to them except via the central node. The local nodes will be required to maintain and populate a core set of data elements which will be termed the global data elements. They can add other data elements (attributes) to the core data as they see fit. The local nodes can develop their own application programs to meet local needs. Each local node will have its own local database administrator (dba).

1.4 Related Work

This paper is related to two areas of distributed database technology. The first area is the design process of distributed databases. The design process for the distributed database in this paper was driven by a specific problem with a specific application in

---

INFORMIX is a registered trademark of INFORMIX Software, Inc.
UNIX is a trademark of AT&T
mind for solving the problem. The second related area is distributed database management functions, specifically, the design of distributed database management algorithms for the management of replicated data. The algorithms in this paper are designed to be implemented on top of INFORMIX-SQL, a relational DBMS.

The initial phase of distributed database design when using a top-down approach (see chapter 2) is, for all practical purposes, the same as designing a centralized database. Centralized database design is normally done with one of four data models in mind; the network, hierarchical, inverted list, or relational. The relational model was used for designing the Financial Control System. The relational model was used because INFORMIX-SQL is a relational DBMS, and the relational model is inherently suited to distribution due to the ease with which it can be horizontally and vertically partitioned [HUSE87]. There is abundant literature on the design of relational databases, the primary sources used by the author were [DATEb86] and [KENT83]. Fragmentation and fragment allocation are areas particular to distributed database design. [CERI84] devotes a chapter to these topics in his textbook on distributed databases. [MOTZ87] presents some algorithms for distribution design which compute optimal fragmentation and allocation based on cost computation and space constraints at each node. [CERI87] presents a top-down distribution design methodology entitled DATAID-D. He also presents a section on related work which succinctly summarizes the important work which has been accomplished in the distributed design area. His paper can be consulted to obtain these sources of related work. The CODASYL Systems Committee Report [CODASY] summarized much of the early work done on the distribution design problem. They concluded that the distribution design problem was often a difficult one and stated: "An additional possibility is to maintain statistics to determine the
actual pattern of usage. This would be useful in the adaptive reassignment of files (fragments) in the network" [CODASY]. All of the literature on distribution design states either explicitly or implicitly that the goal is to place the data in such a way to maximize local access by applications and minimize communication overhead. The Financial Control System achieves this goal with respect to the computation of incurrence summaries (main application program).

The literature on maintenance of replicated data in a distributed database is replete with schemes and algorithms for managing this data. [GARC86] points out that although there is no shortage of proposed algorithms for managing replicated data, these algorithms have seldom been implemented, much less added to commercial products. [LIND87] in describing R*, a DDBMS, states: "In particular we have not implemented support for replicated . . . tables . . . we realized that a major effort would be required to implement such support." [HERL87] surveys some of the related work on management of replicated data. He goes on to present an algorithm which integrates concurrency control and replica management. His algorithm allows trade-offs between concurrency control and availability (replica management). Algorithms such as this are designed for high transaction systems to allow maximum concurrency of transaction processing and increase availability in the face of system failures. The Financial Control System will have few transactions and replication is limited, therefore many of the complex algorithms in the literature do not pertain. [GARC87] states: the simple ideas (algorithms and schemes) are the ones that usually work best in practice. This is especially true in reliable data management, where simpler means less prone to errors and hence more reliable. Since the Financial Control System does not require complex means to manage replicated data, simple replication management schemes will be employed.
The Financial Control System is designed to be a distributed database built on top of a commercial central DBMS (INFORMIX-SQL). The source code for INFORMIX-SQL is not available, therefore certain limitations on application development will be inherent. I found very little in the literature concerning the development of a distributed system on top of a proprietary commercial DBMS. [ZHON87] reports on a system called DdBASE III which is built on top of dBASE III, a commercial database management system. There was very little detail in this article about the actual implementation. [HUSE87] reports on the second year of a multi-year distributed database study. This study is evaluating the applicability of distributed database technology to military command and control systems. The study is being conducted using UNIFY as a component of the distributed system. UNIFY is a commercial DBMS product with proprietary source code. Although this system is more complex than the Financial Control System, many of the problems faced by both systems are very similar.

1.5 Organization of Paper

The paper is organized as follows: Chapter 2 presents the problem which the Financial Control System is being designed to solve. The general requirements for the system are then stated. From this base, the global schema design is described. Chapter 2 finishes with a short description of each global relation in the schema. Chapter 3 presents the distributed part of the database design. This consists of fragmentation and allocation of the global relations. Chapter 4 introduces some problems which will be encountered in the distributed environment. A solution for each of the problems is then presented. The solution to the first two problems incorporates an algorithm which contains a replicated table version control scheme.
and a two phase commit protocol. Chapter 5 concludes the paper with a summary and some observations. The appendix contains a schema for each global relation, the input/output to/from Bernstein's 2nd algorithm, and pseudo-code for the version control and two phase commit algorithm of chapter 4.
CHAPTER 2

DESIGN OF THE GLOBAL SCHEMA

2.1 Introduction

The design of a distributed database usually requires one of two general approaches, the top-down or bottom-up approach. A top-down approach is employed when a distributed database system is being designed from scratch. Conversely, the bottom-up approach is used when existing databases are being aggregated to form a distributed database system [CERI84]. The top-down approach was used for the design of the Financial Control System.

The primary phases of top-down distributed database design are:

- Global schema design
- Fragmentation design
  This phase involves the partitioning of the global schema into logical non-overlapping fragments [CERI84].
- Fragment allocation
  Fragment allocation involves the assignment of the fragments to physical nodes on the network as well as deciding if fragments are to be replicated and where to physically place any replicated fragments.

The design of the global schema is virtually identical to the design of the schema for a centralized database. This chapter will present the design of the global schema for the Financial Control System.
2.2 Analysis of Requirements

Before presenting the global schema design, it may be helpful to describe the problem and the needs of the Ocean Systems Organization of AT&T Technologies' Federal Systems Division. An understanding of the problem and requirements of the organization's management will make certain design decisions much clearer.

2.2.1 The Problem

The Ocean Systems Organization of AT&T Technologies' Federal Systems Division manages various Federal contracts. Tracking incurrences on those contracts is a major function of contract management. Incurrence reports have, in the past, been obtained from a separate accounting organization within Federal Systems Division. The accounting organization's primary function is to precisely account for all incurrences on these contracts to the Government customer. A secondary function of accounting is to report incurrences to the operating and contract management organizations. The contract management and operating organizations were receiving these incurrence reports much too late for them to make any meaningful or timely management decisions such as altering schedules or shifting assets to avoid over-running or grossly under-running budgeted tasks. Various reasons have been attributed to the long lag time involved between submittal of timesheets, vouchers, purchase orders, etc. and the actual reflection of these incurrences in reports. Two primary factors were identified as being responsible for the majority of this lag time. First, the Government imposes certain strict accounting practices on all contractors with large federal contracts. Satisfying these requirements causes delays in the system. Second, accounting has to report incurrences "to the penny". This requirement causes delay in the data entry and reporting of many incurrences.

- 10 -
2.2.2 Output Requirements

To effectively manage incurrences on their government contracts, the Ocean Systems Organization requires three things from a financial control system:

- Timeliness

- Accurate, not necessarily precise, output

- Understandable and meaningful presentation of summary data

2.2.2.1 Timeliness

To effectively manage any type of contract, it is essential to know, in as near real-time as possible, where the enterprise stands in regard to money incurred versus money budgeted. Timeliness of data entry is the most important issue but since this is not something which can be greatly influenced by database design it will not be discussed further. The timeliness issue which can be influenced by the database/application designer is the output of summary reports. Management requires daily updates to summary data so that any summary report will reflect the state of the database from the previous day.

2.2.2.2 Accuracy

To manage the operations of a contract effectively it is not necessary to be able to account precisely for all amounts incurred. The decision on the degree of accuracy is, like timeliness, a subjective matter. No requirements for a specific degree of accuracy have been stated for the Financial Control System. The goal though was to be as close to accounting's figures as possible without overly complicating the database design.
2.2.2.3 Presentation of Output

The requirements for the output of summary or other data is to make it as clear, concise, and understandable as possible. Summary data from the total contract level down to the departmental level should be easily accessed by online users. At each of the vertical levels (departmental, assistant manager, manager), the data is required to be horizontally broken out in incurrence categories. See Table 1.

<table>
<thead>
<tr>
<th></th>
<th>Oct</th>
<th>Nov</th>
<th></th>
<th>Sep</th>
</tr>
</thead>
<tbody>
<tr>
<td>Labor</td>
<td>$</td>
<td>$</td>
<td></td>
<td>$</td>
</tr>
<tr>
<td>Contract Labor</td>
<td>$</td>
<td>$</td>
<td></td>
<td>$</td>
</tr>
<tr>
<td>Purchase Orders</td>
<td>$</td>
<td>$</td>
<td></td>
<td>$</td>
</tr>
<tr>
<td>Travel &amp; Living</td>
<td>$</td>
<td>$</td>
<td></td>
<td>$</td>
</tr>
<tr>
<td>Contractor</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Travel &amp; Living</td>
<td>$</td>
<td>$</td>
<td></td>
<td>$</td>
</tr>
<tr>
<td>Labor Hours</td>
<td>hrs</td>
<td>hrs</td>
<td></td>
<td>hrs</td>
</tr>
<tr>
<td>Contractor</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Labor Hours</td>
<td>hrs</td>
<td>hrs</td>
<td></td>
<td>hrs</td>
</tr>
</tbody>
</table>

**TABLE 1. Incurrence Categories**

The requirements also call for breaking the data down even further into monthly increments for all the vertical organizational levels and horizontal incurrence categories.

2.2.3 Other Requirements

The previous three requirements are related directly to the output from the database. There are other requirements related to the input, integrity and maintenance of the database which are just as, if not more, important. Some of the major requirements here are:

- Ease of data entry or update
- Semantic and integrity constraints
• Application maintenance and development
• Documentation
• User and data entry support

2.2.3.1 Data Entry
The data entry personnel will not necessarily be experienced in data entry or even computer usage. The requirements are therefore to make the data entry, update, and delete functions as straightforward and fool-proof as possible. Custom designed CRT data entry screens which resemble the actual form being entered are required. Menu selection of data entry screens and on-line help are also required. It is envisioned that nearly all data will be entered, updated, and deleted one tuple at a time via a custom screen form. INFORMIX-SQL has a feature called PERFORM for designing these customized screens. Although the majority of data inserted, deleted, and updated will be done one tuple at a time, there is also a requirement for inserting, updating, and deleting multiple rows of data in any of the database tables.

2.2.3.2 Constraints
There are two classes of constraints which should be reflected and hopefully enforced in a relational database. They are semantic and integrity constraints. An enforced semantic constraint would reject data from being entered into a database if that data did not make sense in the databases' real world interpretation. An example from the Financial Control System is a pettycash voucher. In the real world a pettycash voucher cannot exceed an amount of $100. The constraint is enforced if the database rejects the addition or update of a pettycash tuple when the amount is over $100. Such semantic constraints are required in the Financial Control System.
Integrity constraints can be enforced by applying the integrity rules of relational databases. There are three types of relational integrity which are considered in this design:

- Domain
- Entity or key
- Referential

Domain integrity means that the assigned domain of an attribute is enforced. Some examples of domains are integer, money, or date. Values entered into the database must match the format of the domains declared for that attribute.

Entity or key integrity requires that primary key values in base relations must not be null, either in whole or in part [DATEa86].

Referential integrity requires that each foreign key value in a base relation must be either (a) wholly null or (b) equal to the primary key value somewhere within the base relation representing the relevant participant entity. A foreign key is an attribute (or attribute combination) in one relation R2 whose values are required to match those of the primary key of some relation R1 (R1 and R2 not necessarily distinct) [DATEa86].

Enforcement of these integrity constraints is required in the Financial Control System.

2.2.3.3 Application Maintenance and Development

Once the distributed system is implemented, application development and maintenance will be required. It is expected that the database will evolve and grow with time. New applications will have to be developed and old applications
modified, if desired, to take advantage of the evolving database. Maintenance and
development of applications which enforce data consistency and integrity will also be
required.

2.2.3.4 Documentation

Documentation on all phases of design and application development are required.
Documentation should be developed concurrently with the design phase of the
database and during application development.

2.2.3.5 User Support

It is required that users, particularly data entry personnel, be given training for data
entry and use of the system. Data entry personnel should be kept abreast of any
pending modifications to the system which would affect their data entry duties.
Users should be notified of any new system capabilities or any changes which would
affect their view of the database. User or data entry suggestions for system
improvement or requests for other features should be evaluated and acted upon in a
timely manner.

2.3 The Global Schema

Once the problem was defined and general requirements were determined, the design
of the global schema was possible. The schema design had to support a solution to
the problem while attempting to satisfy the requirements. The design of the global
schema proceeded in three distinct phases. The first phase involved identifying and
grouping the data items needed in the database as well as determining dependencies
within the groupings. The second phase, normalization, took the dependencies from
the first phase and produced a third normal form global schema. The third phase,
further normalization, took the third normal form global schema and analyzed it to
see if it required further normalization using a method called decomposition.

2.3.1 Identifying Data Elements

The first step was to identify all of the data elements required in the global schema to support a solution to the problem. Meetings and interviews with contract management personnel provided most of the input needed to select those data items which would be necessary for calculation of incurrences. The remaining data items were identified in talks with personnel experienced in the actual calculations of incurrences.

Once the data elements were identified they were logically grouped into relations. An informal definition of a relation is a table of data items arranged in columns and rows. Each column is defined by a data type or domain. A row is a set of values, one from each column of the table. A formal definition of relation is as follows: A relation on domains D1, D2, ..., Dn (not necessarily all distinct) consists of a heading and a body. The heading consists of a fixed set of attributes (columns) A1, A2, ...., An such that each attribute Ai corresponds to exactly one of the underlying domains Di (i=1, 2, ..., n). The body consists of a time varying set of tuples (rows), where each tuple in turn consists of a set of attribute-value pairs (Ai,vi) (i=1, 2, ..., n), one such pair for each attribute Ai in the heading. For any given attribute-value pair (Ai,vi), vi is a value from the unique domain Di that is associated with the attribute Ai [DATEa86]. An example of a relation for the Financial Control System would be the employee relation made up of the attributes social security number (A1), department (A2), title (A3), and name (A4). The domain of each of these attributes is the ascii character set.

The next step was to identify the functional dependencies among the data elements
in each relation. A functional dependency (FD) is defined as follows: Given a relation R, attribute Y of R is functionally dependent on attribute X of R - in symbols, R.X → R.Y (read "R.X functionally determines R.Y") - if and only if each X-value in R has associated with it precisely one Y-value in R (at any one time). Attributes X and Y may be composite [DATE86].

After the functional dependencies were determined, they were input to a program based on Bernstein's 2nd algorithm [BERN76]. Bernstein's 2nd algorithm produces a relational database schema in third normal form from a set of FDs. The output from this program is located in the appendix. There were thirteen relations in third normal form produced by the algorithm. A fourteenth relation which has no FDs describing the semantics was also added to the global schema. The reason for this will be discussed later.

2.3.2 Normalization

The terms normalized and normal form have been used in the preceding paragraphs without definition. Normalization is the process of converting relations to various levels of normal form based on certain constraints. There are many normal forms. The most familiar being first, second, third, Boyce-Codd, fourth, and projection/join normal form (PJ/NF). Each progressively higher level of normal form is considered more desirable than its' predecessor. More desirable in this context meaning less redundancy in the relation thus avoiding certain update problems associated with redundant data.

An example of an "update problem" associated with a less than optimal normal form will be shown. The "update problem" refers to data insert and delete as well as the update operation. The example relation is called employee(ssn, name, title, dept,
A department can be located in only one city at any one time. The following are some example tuples from the relation:

<table>
<thead>
<tr>
<th>ssn</th>
<th>name</th>
<th>title</th>
<th>dept</th>
<th>dept_loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>111-11-1111</td>
<td>Doe, John A.</td>
<td>SE</td>
<td>1210</td>
<td>LA</td>
</tr>
<tr>
<td>222-22-2222</td>
<td>Young, I. M.</td>
<td>ISA</td>
<td>1210</td>
<td>LA</td>
</tr>
<tr>
<td>333-33-3333</td>
<td>You, I. C.</td>
<td>EA</td>
<td>1211</td>
<td>SF</td>
</tr>
<tr>
<td>444-44-4444</td>
<td>Law, L. A.</td>
<td>ISM</td>
<td>1210</td>
<td>LA</td>
</tr>
</tbody>
</table>

A key of this relation is ssn because it functionally determines the other attributes, in this case it is the only key. The problem is that the dept attribute also functionally determines dept_loc. If department 1210 were moved to NY an update operation would have to access every tuple with dept=1210 and update each dept_loc from LA to NY. This could lead to data inconsistency if the update failed to update every tuple involved. A more desirable design would be to split out dept_loc into a new relation named department.

<table>
<thead>
<tr>
<th>dept</th>
<th>dept_loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1210</td>
<td>LA</td>
</tr>
<tr>
<td>1211</td>
<td>SF</td>
</tr>
</tbody>
</table>

The move to NY would only require the update of one tuple in this case thus eliminating the risk of "update problems".

Normalization theory is a broad field, for those interested in further detail [DATEa86] should be consulted. Suffice it to say that normalization is an important tool to be used in the design of relational database schemas.
2.3.3 Further Normalization

The second part of the normalization process that was conducted is termed decomposition. In decomposition a relation is analyzed to see if certain dependencies are present. If these dependencies are present the relation is non-loss decomposed into two or more relations which do not exhibit the problem dependencies. Non-loss decomposition means that the original relation can be reconstructed from a join of its parts.

The overall objective of normalization is to reduce redundancy in the database. Ultimately, each table should consist of the properties of the key, the whole key, and nothing but the key. Once this stage has been reached the normalization process should stop [DATEb86]. This guideline for practical database design was adopted for the Financial Control System.

The decomposition started with relations in third (or higher) normal form obtained from the Bernstein algorithm. Bernstein’s 2nd algorithm produces a database schema which meets the conditions for third normal form. Many of the relations may already be in Boyce-Codd, fourth, and PJ normal form. Each higher normal form is a sub-set of the previous normal form therefore, a relation in PJ/NF is also in first through fourth normal form. The definition for third normal form is: A relation R is in third normal form (3NF) if and only if the nonkey attributes of R (if any) are a) mutually independent, and b) fully dependent on the primary key of R [DATEa86].

The first step required was to analyze each 3NF relation to see if it required decomposition to Boyce-Codd normal form (BCNF). A relation is in Boyce-Codd normal form if and only if every determinant is a candidate key. A determinant is
any attribute on which some other attribute is (fully functionally) dependent [DATEa86].

Once BCNF was achieved, the decomposition process was halted. At this point, each table met the conditions at which [DATEb86] said further normalization could be stopped. Namely, the relations consisted of a primary key and a set of attributes fully functionally dependent on that key. As it turned out, no decomposition was necessary. All of the relations from Bernstein's 2nd algorithm were already in at least Boyce-Codd normal form. Tables that are in 3NF but not in 4NF or 5NF, though theoretically possible, are very unlikely to occur in practice [DATEb86].

BCNF does not guarantee the removal of all redundancies in the relation but it reduces possible update problems if a relation in 1NF is reduced to a non-loss set of BCNF relations. A counterpoint to the benefits of normalizing to BCNF is that performance of queries in the database may be slower. The reason for this is that extra joins will have to be executed to retrieve elements of data from the decomposed relations. Some balancing of performance against "update problems" will most likely have to be found in practice.

Following are brief descriptions of each global relation. A real world description is given first followed by a list of each functional dependency. At the end of each description is a statement of the relation's normal form.

2.3.3.1 Employee Table

The employee table schema is shown in the appendix. Both AT&T and subcontractor employee records are stored in the employee table. The FDs in employee are:

\[
\text{ssn} \rightarrow \text{name}
\]
Employee is in BCNF because the only candidate key is ssn and it is the only determinant.

2.3.3.2 Mord_nos Table

The schema for the mord_nos table is shown in the appendix. M-order numbers are charge numbers which Ocean Systems employees put on their timesheets to record the hours worked on various tasks. Each m-order number is unique, therefore mor_no is the key of the mord_nos table. M-order numbers from all contracts being tracked are stored in this table. The FDs in mord_nos are:

\[
\begin{align*}
mor\_no \rightarrow & \text{ mor\_title} \\
mor\_no \rightarrow & \text{ cont\_task} \\
mor\_no \rightarrow & \text{ cont} \\
mor\_no \rightarrow & \text{ mor\_type} \\
mor\_no \rightarrow & \text{ fnd}
\end{align*}
\]

Mord_nos is in BCNF because the only candidate key is mor_no and it is the only determinant.

2.3.3.3 Dept_budg Table

The dept_budg table schema is shown in the appendix. Departmental budgets are currently available only down to the contract task level. A contract task can be made up of multiple m-order numbers, therefore direct comparison of incurrences versus budgets at the m-order number level cannot presently be obtained. Contract tasks can span departments, therefore the key is composite and composed of both cont_task and dept. The FDs in dept_budg are:
(dept, cont_task) --> dollar_budg
(dept, cont_task) --> stdhr_budg
(dept, cont_task) --> othr_budg

Dept_budg is in BCNF because the only candidate key is (dept, cont_task) and it is the only determinant.

2.3.3.4 Rates Table

The rates table schema is shown in the appendix. The rates table is the best example of the relaxed requirement for precise accuracy in the output. Individual salaries would have to be maintained to obtain real accuracy. The rates table contains average hourly salary rates at the departmental level. For instance, if a department has three senior engineers in it, the rates table would contain an average senior engineer hourly rate based on the salaries of those three employees. The rates table has rate_start and stop attributes which allow adjustment of individual department rates for any interval of time. The key of the rates table is dept, title, and rate_start. The FDs of rates are:

(dept, title, rate_start) --> dom_hr
(dept, title, rate_start) --> rotdot
(dept, title, rate_start) --> rate_stop

Rates is in BCNF because the only candidate keys (dept, title, rate_start) and (dept, title, rate_stop) are also the only determinants.

2.3.3.5 Spafactor Table

The schema for the spafactor table is shown in the appendix. Spafactor is a small table which records certain adjustments to rates for employees working night shift, overseas locations, or at sea. If an employee’s timesheet has a non-zero entry in the
spa field, the spafactor table is referenced to obtain the proper adjustments to the
calculation of labor dollars. The FDs in spafactor are:

\[
\begin{align*}
\text{spa_code} & \rightarrow \text{code_def} \\
\text{spa_code} & \rightarrow \text{spa_factr} \\
\text{spa_code} & \rightarrow \text{spa_dollars}
\end{align*}
\]

Spafactor is in BCNF because the only candidate key is spa_code and it is also the
only determinant.

2.3.3.6 Timsheet Table

The timsheet table schema is shown in the appendix. The timsheet table holds the
pertinent information from each employee's bi-weekly timesheet. The key is
composite and consists of ssn and enddate. An employee records each m-order
number with associated hours worked on the timesheet. If more than one m-order
number is worked during the two week time period, the primary number used will be
entered in the main.mo field of the timsheet table and the other m-order numbers
will be entered in the next table to be described. Timsheet contains the following
FDs:

\[
\begin{align*}
(\text{ssn, enddate}) & \rightarrow \text{dept} \\
(\text{ssn, enddate}) & \rightarrow \text{main_mo} \\
(\text{ssn, enddate}) & \rightarrow \text{ot_total} \\
(\text{ssn, enddate}) & \rightarrow \text{tpnw} \\
(\text{ssn, enddate}) & \rightarrow \text{spa} \\
(\text{ssn, enddate}) & \rightarrow \text{sea_days}
\end{align*}
\]

Timsheet is in BCNF because the only candidate key (ssn, enddate), is the only
determinant.
2.3.3.7 Tim_detail Table

The tim_detail schema is shown in the appendix. As mentioned in the previous section, any m-order numbers worked, other than the main m-order number, are recorded in the tim_detail table. For any unique timsheet record there may be zero, one or more corresponding tuples in the tim_detail table. The key of tim_detail is composed of ssn, enddate, and mor_no. The FDs in tim_detail are:

\[(ssn, enddate, mor_no) \rightarrow stdhrs\]
\[(ssn, enddate, mor_no) \rightarrow othrs\]
\[(ssn, enddate, mor_no) \rightarrow chrgdept\]
\[(ssn, enddate, mor_no) \rightarrow spa\]
\[(ssn, enddate, mor_no) \rightarrow sea_days\]

The tim_detail relation is in BCNF because the only candidate key is also the only determinant.

2.3.3.8 Payperiod Table

The payperiod table schema is shown in the appendix. The payperiod table is closely associated with the timsheet table. It contains the number of weekday hours available in each bi-weekly time period. This table is used to look-up the number of hours worked on the main m-order number so the data entry people do not have to calculate it at data entry time. The FD in payperiod is:

\[enddate \rightarrow per\_length\]

Payperiod is in BCNF because the only candidate key (enddate), is also the only determinant.

2.3.3.9 Cont_time and Cont_detail Tables

The schemas for these two tables are shown in the appendix. These two tables are
analogous to the timesheet and tim_detail tables. They are designed to record the timesheets of sub-contractor employees. The major difference between the cont_time and timesheet tables is that sub-contractor’s time reporting periods can be variable. Cont_time has tstart and tstop attributes to record this. The primary key of cont_time is ssn and tstart. The FDs of cont_time are:

\[(ssn, tstart) \rightarrow tstop\]
\[(ssn, tstart) \rightarrow dept\]
\[(ssn, tstart) \rightarrow main_no\]
\[(ssn, tstart) \rightarrow stdhrs\]
\[(ssn, tstart) \rightarrow othrs\]
\[(ssn, tstart) \rightarrow tpnw\]
\[(ssn, tstart) \rightarrow spa\]

Cont_time is in BCNF because the candidate keys \((ssn, tstart)\) and \((ssn, tstop)\) are also the only determinants.

The key of cont_detail is composed of ssn, tstart, and mor_no. The FDs in cont_detail are:

\[(ssn, tstart, mor_no) \rightarrow stdhrs\]
\[(ssn, tstart, mor_no) \rightarrow othrs\]
\[(ssn, tstart, mor_no) \rightarrow chrgdept\]
\[(ssn, tstart, mor_no) \rightarrow spa\]

Cont_detail is in BCNF because the candidate key \((ssn, tstart, mor_no)\) is also the only determinant.

2.3.3.10 Voucher Table

The voucher table schema is shown in the appendix. A voucher is used to record
travel and living expenses on an employee’s business trips. The key is composed of ssn and vouch_no (an employee generated sequence number for his/her vouchers).

The FDs in voucher are:

\[(ssn, vouch\_no) \rightarrow mor\_no\]
\[(ssn, vouch\_no) \rightarrow amount\]
\[(ssn, vouch\_no) \rightarrow vouch\_date\]
\[(ssn, vouch\_no) \rightarrow chrgdept\]

The voucher table is in BCNF because the only candidate key (ssn, vouch_no), is also the only determinant.

2.3.3.11 Purch\_ord Table

The schema for the purch\_ord table is shown in the appendix. Purchase order forms are used by the organization for purchasing materials and services. Purchase orders are applied on m-order numbers and charged to the department ordering the materials or services. The purch\_ord FDs are:

\[purch\_ord\_no \rightarrow req\_purch\]
\[purch\_ord\_no \rightarrow order\_date\]
\[purch\_ord\_no \rightarrow mor\_no\]
\[purch\_ord\_no \rightarrow dept\]
\[purch\_ord\_no \rightarrow amount\]

The purch\_ord relation is in BCNF because the only candidate key, purch\_ord\_no, is also the only determinant.

2.3.3.12 Pettycash Table

The schema for the pettycash table is shown in the appendix. The pettycash form is used to account for miscellaneous expenses which do not exceed $100. The key is
composed of ssn, and a sequence number(emp_seq_no). The FDs in pettycash are:

(ssn, emp_seq_no) -> mor_no
(ssn, emp_seq_no) -> gn250_date
(ssn, emp_seq_no) -> amount
(ssn, emp_seq_no) -> chrgdept

Pettycash is in BCNF since the candidate key (ssn, emp_seq_no), is the only determinant.

2.3.3.13 GN300 Table

The GN300 table schema is shown in the appendix. The GN300 form is used to report the spending of money on items which do not fall under the categories of travel and living, purchase of materials or services, or pettycash. The GN300 form does not have fields on it which could be used as a unique key in a relation. Since there are very few of these forms generated, it was decided not to generate a fictional unique attribute to be the key. The effect of this decision is that duplicate tuples can exist in the table and they can cause update and delete problems. These problems are recognized and will be monitored. Another effect is that any search of the table has to be a serial versus an indexed search. Since the number of tuples stored in this table is small, and expected to stay that way, this should not be a problem.

2.4 Evaluating the Global Schema

The global schema design satisfies some of the requirements stated earlier. Specifically, it has provided the data elements required to calculate incurrences accurately. It impacts ease of data entry by having identified only those data elements pertinent to the solution of the problem. It eases data maintenance (update, delete) by reducing redundancy. The global schema represents entity
integrity through the identification of FDs. It supports the development of powerful queries (applications) across multiple relations through the inclusion of foreign keys. Finally, the global schema by itself is a form of documentation.

Other requirements such as timeliness of data entry and report generation, output presentation, custom data entry screens, domain and referential integrity, documentation, and user support cannot be met by global schema design. These requirements will have to be satisfied by application programs or other means.

The design process used for each relation in the global schema has attempted to capture the semantics of the real world with as little redundancy as possible. Many of the real world semantics such as the pettycash example given earlier cannot be enforced in schema design but can be addressed via the design of custom data entry screens or other application programs.

Entity or key integrity has been designed into each base table except for the GN300 table mentioned earlier. Entity integrity is enforced in INFORMIX-SQL by explicitly creating a unique index for each primary key as well as specifying that no attribute which is a part of the primary key can receive null values.

Domain integrity is supported by INFORMIX-SQL through its column type feature.

There are numerous foreign keys designed into the global schema. These foreign keys allow "navigation" thru the database and provide the common attributes around which relational operations such as join are constructed. The referential integrity of these foreign keys is not fully supported by INFORMIX-SQL. Referential integrity can be enforced via the custom data entry screen but it is not enforced if values are bulk loaded from an ascii file. Applications will have to be developed to enforce full referential integrity.
2.5 Summary Table

The summary table schema is displayed in the appendix. The summary table is not a base relation in the global schema. It is a derived table which is created and populated by an application program. The summary table will be the source for most of the reports and on-line queries in the database. It will require one additional attribute(month) for incurrences to be broken out by monthly time periods.
CHAPTER 3
DISTRIBUTION DESIGN

3.1 Introduction

Fragmentation is the process of subdividing a global object (entity or relation) into several pieces called fragments. Allocation is the process of mapping each fragment to one or more nodes. The combination of fragmentation and allocation design can be termed distribution design. A key principle in distribution design is to achieve maximum locality of data and applications [CERI87].

3.2 Fragmentation

Fragmentation of a relation can be accomplished by two methods, horizontal or vertical fragmentation.

3.2.1 Horizontal Fragmentation

Since the Financial Control System is a relational database, I will use some relational algebra terminology to describe horizontal and vertical fragmentation. Horizontal fragmentation of a global relation is achieved by applying the selection operation of the relational algebra. A selection predicate is required to obtain a subset of the global tuples. The selection predicate contains the value of an attribute, or attributes, from the relation. An example of a horizontal fragmentation of a relation, say emp(ssn, name, dept), using SQL is as follows:

```
select ssn, name, dept from emp
where dept = 1325
```

This would produce a subset of tuples from the emp relation containing only those employees in department 1325. The selection predicate in this example is the value
of the dept attribute (1325). If there are a number of different departments in the
global relation, the selection operation can be repeatedly executed with a new
department to produce a set of disjoint horizontal fragments. The global relation can
be reconstructed by applying the union operation of the relational algebra.
The rationale of horizontal fragmentation is to produce fragments with the
maximum potential locality with respect to operations, i.e., such that each fragment
is located where it is mostly used [CERI87].

3.2.2 Vertical Fragmentation
Vertical fragmentation of a global relation is achieved by applying the projection
operation of the relational algebra. The projection operation requires no selection
predicate. It merely requires a subset of the global relation's attributes as input.
Using the previous relation a projection over that relation in SQL would be:

```
select ssn, name from emp
```
This operation would result in a fragment composed of all social security numbers
and corresponding names from the emp relation. All vertical fragments are required
to have as members the key attribute(s) of the global relation so that they can be
reconstructed (without loss) by applying the join operation of the relational algebra.
The rationale of vertical fragmentation is to cluster attributes frequently used
together. An ideal vertical fragmentation exists when each application uses just one
subset of attributes; otherwise, some applications will be harmed, since they will need
to access both fragments. In this general situation, one has to balance potential
benefits (due to the possibility of placing each fragment close to the applications
which mostly use it) against potential disadvantages (due to the same applications
accessing two fragments) [CERI87].
It should be clear from the above discussion that a good understanding of the most important applications is needed before informed decisions can be made regarding fragmentation.

3.3 Allocation

Once a fragmentation design has been achieved, the next step is to physically place those fragments on nodes of the network. At this point, a decision has to be made on the degree of replication desired in the design. The choices in this phase range from full replication to no replication.

3.3.1 Full Replication

An allocation design with full replication has no fragmentation. Each node of the distributed database contains a copy of the global database. The primary advantage of this situation is the availability obtained. The entire database is available while at least one node is active. The primary disadvantage of this situation is that any change to the database has to be propagated reliably to all nodes in order to maintain data consistency and integrity.

3.3.2 No Replication

The other end of the spectrum is a totally disjoint database, i.e. no replication. The data from each node would have to be joined via union to obtain a single copy of the global database. The primary advantage of this design is that there is no overhead required to maintain consistency and integrity of replicated data. The main disadvantage is that availability of the system is greatly reduced. The loss of one node will create gaps in the global data.

The optimal allocation design will normally fall somewhere in between these two extremes.
3.4 Distribution Design of the Financial Control System

While distributed databases enable more sophisticated communication between sites, the major motivation for developing a distributed database is to reduce communication by allocating data as close as possible to the applications which use them. However, it rarely occurs that data and applications can be cleanly partitioned and assigned to a particular site [CERI87]. Fortunately, the Financial Control System is one of those rare cases where the organizational structure of the corporation and the nature of the main application creates a "natural" distribution.

The bulk of raw data to be stored in the database consists of employee timesheets and vouchers. Since all employees use the same timesheet and voucher forms, it allows a natural horizontal fragmentation of timesheets and vouchers at the assistant manager level. The other relations which contain the department attribute were likewise fragmented on the department attribute at the assistant manager level. The fragmentation is done at the assistant manager level to aggregate enough employees to justify the hardware and software required to support a local node. Using SQL, a horizontal fragment of the timsheet relation would be defined like this:

```
select timsheet.* from timsheet
where dept matches "135*"
```

This statement would select all tuples from the timsheet relation where the employee's department was 1351 thru 1359. The assistant manager's are designated as 1350, 1360, 1320, etc., therefore this SQL statement would define a horizontal fragment at the 1350 assistant manager level.

The allocation of these fragments falls naturally in place after the fragmentation is decided. Each assistant manager database (local node) will contain the raw data of...
it's own departments.

The remaining question left is the amount of replication needed. The Financial Control System is not a high transaction system with strict requirements on currency of the data. The loss of a node for a short period of time would not harm the utility of the system to any great extent. For these reasons, there is no replication of any timesheet, voucher, or other employee related raw data. The only relations replicated are those controlled by the central node. These relations experience minimal insert, update, and delete activity. Replicating these relations will therefore require minimal overhead for maintaining consistency of the replications. These relations were fragmented and allocated to the local nodes primarily to place the data where the main application needs it.

The following table presents the distribution design.

<table>
<thead>
<tr>
<th>Table</th>
<th>Control</th>
<th>Fragmentation</th>
<th>Replication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee</td>
<td>Local Node</td>
<td>Horizontal</td>
<td>None</td>
</tr>
<tr>
<td>Timsheet</td>
<td>Local Node</td>
<td>Horizontal</td>
<td>None</td>
</tr>
<tr>
<td>Tim_detail</td>
<td>Local Node</td>
<td>Horizontal</td>
<td>None</td>
</tr>
<tr>
<td>Cont_time</td>
<td>Local Node</td>
<td>Horizontal</td>
<td>None</td>
</tr>
<tr>
<td>Cont_detail</td>
<td>Local Node</td>
<td>Horizontal</td>
<td>None</td>
</tr>
<tr>
<td>Mord_nos</td>
<td>Central Node</td>
<td>None</td>
<td>Full</td>
</tr>
<tr>
<td>Dept_budg</td>
<td>Central Node</td>
<td>Horizontal</td>
<td>Partial</td>
</tr>
<tr>
<td>Rates</td>
<td>Central Node</td>
<td>Horizontal</td>
<td>Partial</td>
</tr>
<tr>
<td>Spafactor</td>
<td>Central Node</td>
<td>None</td>
<td>Full</td>
</tr>
<tr>
<td>Voucher</td>
<td>Local Node</td>
<td>Horizontal</td>
<td>None</td>
</tr>
<tr>
<td>Purch_ord</td>
<td>Central Node</td>
<td>Horizontal</td>
<td>Partial</td>
</tr>
<tr>
<td>Pettycash</td>
<td>Local Node</td>
<td>Horizontal</td>
<td>None</td>
</tr>
<tr>
<td>GN300</td>
<td>Local Node</td>
<td>Horizontal</td>
<td>None</td>
</tr>
<tr>
<td>Payperiod</td>
<td>Local Node</td>
<td>None</td>
<td>Full</td>
</tr>
</tbody>
</table>

**TABLE 2. Distribution Design**

Control means that the node has read/write permission versus read permission only.

Horizontal Fragmentation with Replication of None indicates that there are disjoint fragments of the relation at each local node, the central node has no copy of this
data. Horizontal Fragmentation with Replication of Partial indicates that the global relation resides at the central node and horizontal fragments reside at each local node. Fragmentation of None and Replication of Full indicates that the global relation resides at every node.
CHAPTER 4

ALGORITHMS FOR IMPLEMENTING
CONSISTENCY OF REPLICATED DATA

4.1 Introduction

Five problem areas in maintaining consistency of distributed replicated data have been defined and either an algorithm or some other type of solution has been designed to solve each problem. The problems are:

1) For the replicated tables controlled by the central node, how will the system ensure that each local node has the most current version at any one time?

2) It is required that any transactions (update, insert, delete) on the replicated tables by the central node either all commit or all abort (atomicity). Additionally, any of these transactions must be correctly completed in the event of communication or node failures. Therefore the problem is: how will the system ensure the atomicity of the transactions on the replicated tables?

3) If a local node receives an update to a table controlled by the central node while it is running a summary report, what action must the local node take?

4) If a local node receives an update to a table controlled by the central node after it has just completed a summary report but before it has sent the report to the central node, what action must the local node take?

5) If a local node or the central node discovers that the summary report data sent to the central node is in error, what action must be taken?

4.2 Problems 1 and 2

Problems 1 and 2 both deal with the management of replicated data. All of the
replicated data in the Financial Control System is maintained by the central node. The replicated tables are:

```
depth_budg
rates
spafactor
purch_ord
mord_nos
payperiod
```

Only the central node can add or change data in these tables, the local nodes may only read the data in these tables. Updates, inserts, and deletes of tuples in these tables is coordinated from the central node.

### 4.2.1 Replicated Data

A major reason for replicating data in a distributed system is to increase performance. This is precisely why data was replicated in the Financial Control System. Each local node has all of the data it needs to locally execute the summary report. A price has to be paid for the advantage of having this data available locally. This price is the cost of updating all replicated tables and ensuring the consistency of the data in those tables. Maintaining the consistency of this replicated data is very important for the calculation of accurate incurrence figures. The Ocean Systems Organization has placed emphasis on maintaining the consistency of replicated data to ensure the best possible incurrence figures. In the following sections a version control scheme and a two phase commit protocol will be presented. Both of these are designed to maintain the consistency of replicated data in the Financial Control System.

### 4.2.2 Requirements, Assumptions, and Definitions

To implement the algorithm, the following are required:

1) Transaction logging is required at all nodes. The transactions must be written to a
stable storage device.

2) A remote procedure call (RPC) facility is required at each node.

3) Any concurrency control needed at a local node must be supplied by the local INFORMIX-SQL DBMS.

When designing an algorithm dealing with complex systems, certain simplifications are usually made to keep the algorithm from becoming overly complicated. If the Financial Control System's algorithm was designed to attempt recovery from every conceivable node or communication failure, it too would become overly complex. Therefore, the algorithm assumes a model of "well behaved" system failures. Well behaved means communication failures consist only of lost messages or timeouts and node failures are clean. Clean meaning a node is either active or failed and a failed node is easily detected. In addition, all failures are assumed to be hardware related, the software is assumed to work without error.

The following are definitions for some of the terms found in the algorithm:

PERFORM screen is a feature of INFORMIX-SQL which facilitates the design of custom screens to be used for database query, insert, update, or delete. PERFORM allows these actions on one tuple at a time and provides automatic locking of the tuple when the update option is chosen. It also provides a facility for passing tuple values to an ESQL/C program.

ESQL/C is embedded SQL for C programs. Standard SQL statements can be included and compiled in a standard C program.

Client and server are names given to the calling and called procedures respectively in the remote procedure call model.
4.2.3 Supporting Relations Required

Some additional relations will be required to support the solutions to problems 1 and 2. These relations will be controlled and maintained by the central node.

At Central Node:

rep_tbps table:
- mord_nos: yes
- rates: no
- dept_budg: no
- spafactor: yes
- purch_ord: no
- payperiod: yes

net_addr table:
- local_nodid: 1320
- addr: 1320
- db_name: gcuxh
- results
- local_nodid: 1340
- addr: 1340
- db_name: gcsql
- finance
- local_nodid: 1350
- addr: 1350
- db_name: gcato
- results
- local_nodid: 1360
- addr: 1360
- db_name: port
- incurred

version_tbl table:
- local_nodid: 1320
- table: mord_nos
- version: 6
- table: rates
- version: 27
- table: dept_budg
- version: 52
- table: spafactor
- version: 75
- table: purch_ord
- version: 92
- table: payperiod
- version: 1

At Local Nodes:

version_tbl table:
- local_nodid: 1320
- table: mord_nos
- version: 6
- table: rates
- version: 27
- table: dept_budg
- version: 52
- table: payperiod
- version: 1

TABLE 3. Supporting Relations for Algorithm

Table 3 shows only the 1320 fragment at the 1320 node. The fragments at the other local
nodes would differ only on the values of the local_nodid and version attribute.

4.2.4 The Algorithm

The version scheme and the two phase commit protocol are incorporated into one algorithm. This is done to ensure that any update, insert, or delete to any replicated table is done on the latest version of that table. The version control scheme was an original development for the Financial Control System. Theoretically, with the two phase commit protocol used here, the versions of a replicated table should never disagree. The version control scheme adds assurance that, should a failure of some type leave a disagreement in table version numbers, the discrepancy will be corrected. The two phase commit protocol is described in numerous books and papers. I adapted the protocols found in [CERI84], [GARC87], and [MAEK87] to fit the Financial Control System. The two phase commit protocol used here is designed to install its transaction at all nodes or none at all. In the following algorithm I will describe an update transaction; an insert or delete requires essentially the same actions.

Central Node's Actions:

1) A PERFORM screen is activated and the tuple to be updated is brought up on the screen.

2) The update option is chosen and the attribute to be updated is changed on the screen.

3) The above action calls an ESQL/C program (the client) which will be responsible for propagating the update to the local node(s).

4) The ESQL/C program is passed the old and new value of the updated attribute(s), the key attribute's value, the dept attribute's value (if applicable), and the table
name.

5) Using the table name and the dept, the network address(es) and database name(s) are obtained from rep_tbl and net_addr tables.

6) With the table name and network address(es), the version table is queried to obtain the current version number of the table at that address. Since updates are infrequent the version number will be a number between 1-99 and will recycle when it reaches 99.

7) The transaction is started (BEGIN WORK), pertinent information is written to the transaction log - old and new update value, network address, version number, table name, database name, update attribute name.

Phase I:

8) With the local node's address, a remote procedure call (RPC) is made to the local node(s) passing the local database name, the table name, the key attribute and value, the update attribute and values (old and new), and the current version number.

9) A timeout is activated and the procedure blocks while waiting for a response from the local node(s).

10) If any node sends back an abort, a global_abort is written to the transaction log and an abort message is sent to the local node(s). Next, wait for ack_abort from local node(s), and when they are received, ROLLBACK WORK. The transaction is restarted a second time.

a) If an abort with an old version number i.e. (abort_bad_ver 26) is received from a local node, a global_abort is written to the transaction log and an abort message is sent to the local node(s). Again, wait for ack_abort from local node(s), and ROLLBACK WORK. Call version update procedure.

b) The version update routine will propagate the missing transaction(s). At this
point write message to PERFORM screen - "Updating Old Version" so that the person doing the update will realize a short delay will be experienced. The version update procedure is passed the addr, the table and dbname, as well as the most recent table version number and the old version number from the local node. The transaction log at the central node is accessed and all missing transactions for the table are extracted. These transactions are installed at the local node using a one phase commit protocol. The local node acks the installation of the transaction(s) and the version update procedure restarts the original transaction.

c) If all nodes report Ready, start Phase II.

Phase II:

11) Increment version table's version value by 1.

12) Write global_commit to log, send a COMMIT message to the local node(s).

13) Wait for ack_commit from local node(s). On receipt of them COMMIT WORK.

End of 2 phase commit protocol for central node. Return to PERFORM screen and report "Record Updated."

Local Node's Actions:

1) The server process is activated on receipt of the remote procedure call from the client process (central node).

2) The database is selected.

3) Get table name and version number from central node's RPC data. Query local version table to see if version numbers match. If so, start Phase I.

   a) If version numbers do not match send abort_bad_ver and local version number to central node. Wait for global_abort. Send ack_abort to central node and exit.
b) Note: The central node will now activate a version update procedure which will install the missing transaction(s) and then restart the original transaction.

Phase I:

4) The transaction is started (BEGIN WORK). The transaction log is written with the pertinent update information. The update is executed. Increment version value of version table by 1.

5) If any errors detected, send abort to central node.

6) If update was successful, write Ready in log and send Ready to central node.

Phase II:

7) If an abort is received, ROLLBACK WORK and send ack_abort to the central node.

8) If COMMIT message is received, execute COMMIT WORK and send an ack_commit to the central node. End of 2 phase commit protocol for local node.

9) If transaction was aborted by us, send msg to local dba of that fact.

Note:

A transaction will be attempted two times before the central node will abort and quit trying. The ESQL/C program will return a message to the PERFORM screen saying "Update Aborted by Node(s): xxxx xxxx"

A pseudo-code program listing for the above algorithm is contained in the appendix.

4.2.5 Algorithm Structure

Figure 2 shows the activities of the central and local nodes during a successful version control and two phase commit for an update transaction.

Figure 3 shows the structure of the algorithm in relation to the activities at the central and local nodes. The numbers in the figure correspond to the numbered
actions at the central and local nodes described in the previous section.

Figure 2. Replicated Table Update Transaction
Central Node:

Database

Client Procedure ESQL/C Program

Transaction Log

Local Node:

Server Procedure ESQL/C Program

Transaction Log

Local Database

Figure 3. Algorithm Structure
Figure 4. Financial Control System Structure

Figure 4 shows an overview of the entire Financial Control System. It graphically depicts the components involved in a distributed insert, update, or delete to the replicated tables.

4.2.6 Failure Recovery

To preserve consistency a commit protocol must have a recovery algorithm to ensure a transaction is completed properly after a failure has been experienced.
4.2.6.1 Node Failures

A node failure can include any of the local nodes as well as the central node.

4.2.6.1.1 Failure One

A local node fails before having written Ready in the log. In this case, the central node’s timeout expires, and it takes the abort decision. All active local nodes abort their transactions. When the failed local node recovers, the recovery procedure aborts the transaction.

4.2.6.1.2 Failure Two

A local node fails after having written Ready in the log and sending Ready to the central node. In this case the active local nodes correctly terminate the transaction (commit or abort). When the failed node recovers, the recovery procedure has to ask the central node what the outcome of the transaction was. The transaction is then correctly completed by the local node.

4.2.6.1.3 Failure Three

The central node fails after having written the update data to the log and sending this data to the local nodes. In this case all local nodes which have answered Ready must wait for the recovery of the central node. The recovery procedure of the central node resumes the commitment protocol from the beginning, reading the identity of the local nodes from the transaction log. Each ready local node must recognize that the new data is a repetition of the previous data.

4.2.6.1.4 Failure Four

The central node fails after having written a global_commit or global_abort record in the log, but before having written COMMIT WORK in the log. In this case, the central node at restart must send to all local nodes the decision again. All local
nodes which have not received the **commit** or **abort** commands must wait for recovery of the central node. Again, local nodes should not be affected by receiving the command message twice.

4.2.6.1.5 Failure Five

The central node fails after having written the **COMMIT WORK** record in the log. In this case, the transaction has been concluded and no action is required at recovery.

4.2.6.2 Lost Messages

Lost messages mean completely lost. No provisions for receipt of garbled messages are provided in this algorithm design.

4.2.6.2.1 Lost Message One

A reply message (Ready or **abort**) from a local node is lost. In this case the central node’s timeout expires, and the transaction is aborted. This failure is observed only by the central node, and from the central node’s viewpoint it is exactly like the failure of a local node. From the local node’s viewpoint the situation is different; the local node does not consider itself failed and does not execute a recovery procedure.

4.2.6.2.2 Lost Message Two

The initial transaction data to the local node is lost. In this case the local node’s server process is not activated. The global result is the same as in the previous case, because the central node does not receive a reply.

4.2.6.2.3 Lost Message Three

A command message from the central node (**commit** or **abort**) is lost. The local node remains uncertain about the decision. A timeout in the local node would solve this problem. If no decision has been received after the timeout interval, the local node
requests a repeat of the transaction decision.

4.2.6.2.4 Lost Message Four

The final ack (ack_abort or ack_commit) message is lost. The central node is uncertain whether or not the local node has received the command message. This problem can be eliminated by introducing another timeout in the central node. If no ack message is received after the timeout interval from sending the command message, the central node sends the command again. The best action at the local node is to send the ack message again, even if the transaction was completed and is no longer active.

The above recovery procedures were all adapted from procedures found in [CERI84].

4.3 Problem 3

The solution to problem 3 is based on the following assumption.

Assumption: Ocean System’s management has decided that an update from the central node takes precedence over the compilation of a summary report.

Working under this assumption, any transaction initiated by the central node at any local node will terminate an active summary report compilation. This can be accomplished quite easily by adding a couple of steps to the previous algorithm. In the local node’s actions add two new steps:

1) A check is made to see if a summary report is currently executing. If so it is killed. The summary report will trap the kill signal and die gracefully.

This step is added between the present step 1 and step 2.

2) If the summary report was killed, restart the summary report and send a message to the local dba notifying him/her that the summary was killed and restarted.

This step is added at the end of the present algorithm for the local node.
After any active summary report is killed at a local node, the central node continues on with it's version control and two phase commit protocol.

4.4 Problem 4

Problem 4 rephrased: A local node has just completed a summary report. An update, insert, or delete is successfully transacted on one of the replicated tables. The summary report has not been sent to the central node. What action should be taken?

A solution to this problem is based to some extent on the assumption for problem 3. It is desired to run summary reports with the most up to date input data. If the above situation is encountered, it can be handled in the following manner. Summary reports are sent via uucp to the central node. When a summary report successfully completes, it automatically sends the summary data. The uucp job number can be stored in a file. After a change to a replicated file has been installed, a utility program can be called which will read the last line in this summary report uucp file. A comparison of this data with the output of the uustat (status of uucp jobs) command will tell if the summary report is still in the uucp queue. If the job is in the queue, it can be cancelled and the summary report called and executed again.

4.5 Problem 5

Problem 5 requires that there be some mechanism to identify and remove a local node's summary report data from the central node's aggregated summary data. This would be neccessary in the event that bad summary data was sent from a local node and aggregated at the central node with the data from the other local nodes.

The present design does not have an efficient method to do this. Once the local
node's summary data is aggregated by the central node there is no way to identify a
tuple as having been originated from any particular node. To back out the incorrect
data would require each node to re-calculate and re-submit the summary data.

A solution to this is to add an attribute to the summary table which would contain
the value of the local node's identification i.e. 1320, 1350, 2040, etc. Now every
tuple in the summary table has its source incorporated. When the dba of a local
node discovers that incorrect summary data has been generated, he/she can notify
the central node's dba of this fact and the affected tuples can be deleted from the
summary table.

The solutions to problems 3, 4, and 5 are all original. They are designed to solve
specific problems expected in the Financial Control System. There were no similar
problems or solutions found in any of the literature I read.
CHAPTER 5
SUMMARY

5.1 Introduction

A design for a distributed database and a distributed application for that database has been presented. This system is different from most in the literature by virtue of being built on top of a commercial centralized database management system. Other distributed database management systems have been designed and implemented as extensions to centralized DBMSs (Distributed Ingres and R* being two of the most well known [LIND87]).

5.2 Simplifications

There are limitations to the DDBMS functions that can be implemented when a system is being built on top of a commercial product without source code available. Certain simplifications have to be made under these circumstances. The first simplification for the Financial Control System was to design only those distributed management functions required by the main application. These functions included a centrally coordinated global update, insert, and delete feature with a two phase commit protocol. A simple consistency scheme (version control) was also designed. The sharing of global data is provided through the propagation of summary incurrence reports. Some of the important distributed database management features not offered in this design are:

1) Global queries
2) Global concurrency mechanisms

Local node concurrency (using a locking mechanism) is available through
INFORMIX-SQL.

3) Distributed database administration functions

Local database administration functions will be provided at each node by
INFORMIX-SQL.

5.3 Overview

The evolution of the Financial Control System has gone from the recognition of a
contract management problem to the present design of a distributed system to solve
that problem. During this process the following actions have been performed:

1) The problem was defined.

2) The requirements for a solution were defined.

3) The data elements required to support a solution were identified.

4) The data elements were grouped into entities.

5) The relationships between entities were analyzed.

6) The functional dependencies within entities were determined.

7) The functional dependencies were input to Bernstein’s 2nd algorithm and a
synthesized set of relations in 3NF were output.

8) The relations and FDs were analyzed to see if any relation was not in BCNF.

9) The global database schema was composed.

10) The fragmentation design of each global relation was decided.

11) Fragments were allocated to database nodes.

12) Some application dependent problems were identified.

13) Solutions to these problems were designed.

There were a number of constraints kept in mind at each step in the design process.

The most important of these were:
1) integrity
2) local node autonomy
3) summary report requirements
4) consistency
5) keep it simple (therefore reliable)

5.4 Lessons Learned

More time should have been devoted to researching and selecting the data items for the global schema. There were some relations and attributes added late in the design stage which had been overlooked. The pettycash and gn300 relations are two examples. Iteration of design is to be expected but the importance of careful and meticulous research at the global schema design stage cannot be over-emphasized. Relational databases are quite forgiving in this regard since most applications will still work properly after a new relation or an attribute is added to the database. An application would only have to be modified to incorporate this new data. Given this forgiveness, it is still very important to capture the best possible representation of the real world semantics as early as possible. This was difficult in the design of the Financial Control System due to the lack of an expert with a broad understanding of contract incurrence computations.

The two phase commit protocol presented in this report is known as the centrally coordinated two phase commit. The algorithm for this protocol allows concurrent processing of a transaction at the local nodes. If another protocol were to be evaluated, it would be worthwhile to take a serious look at the linear (nested) two phase commit protocol. The communication topology for this protocol is a linear chain. The coordinator (central node) sends the data to the first local node in the
chain, this node then decides ready or abort. If the decision is abort, the central node is notified of the abort. If the decision is ready, then the next local node in the chain is passed the data and it then decides ready or abort. This continues, while the decision is ready, until the last node in the chain is reached. If the transaction reaches the last node, this means that all the previous nodes are ready to commit. At this stage, the last node becomes the coordinator and, based on its decision, it passes either commit or abort back through the chain of local nodes to the central node. This protocol forfeits the concurrency of the centrally coordinated protocol for lower communication overhead. A successfully committed transaction requires 4n messages (where n is the number of nodes) with the centrally coordinated protocol. The linear protocol requires 2n messages for a successfully committed transaction. The linear protocol is appropriate for a system with the following characteristics:

1) There is a high cost with message passing and a broadcast facility is not available.
2) The demand for concurrency is low.
3) The cohort (local node ESQL/C program) structure is static or universally known [MAEFK87].
REFERENCES


KENT83 A simple guide to five normal forms in relational database theory. Kent, W. ACM Communications 26, 2, (Feb 1983) 120 - 125.


APPENDIX

***** Bernstein's 2nd Algorithm Input/Output *****

THE INPUT TO THE PROGRAM IS:
EMPLOYEE.SSN > EMPLOYEE.NAME;
EMPLOYEE.SSN > EMPLOYEE.TITLE;
EMPLOYEE.SSN > EMPLOYEE.DEPT;
MORDNOS.MORNO > MORDNOS.MORTITLE;
MORDNOS.MORNO > MORDNOS.CONTTASK;
MORDNOS.MORNO > MORDNOS.CONT;
MORDNOS.MORNO > MORDNOS.MORTYPE;
MORDNOS.MORNO > MORDNOS.FND;
DEPTBUDG.DEPT, DEPTBUDG.CONTTASK > DEPTBUDG.DOLLARBUDG;
DEPTBUDG.DEPT, DEPTBUDG.CONTTASK > DEPTBUDG.STDHRBUDG;
DEPTBUDG.DEPT, DEPTBUDG.CONTTASK > DEPTBUDG.OTHRBUDG;
RATES.DEPT, RATES.TITLE, RATES.RATESTART > RATES.DOMHR;
RATES.DEPT, RATES.TITLE, RATES.RATESTART > RATES.ROTDOT;
RATES.DEPT, RATES.TITLE, RATES.RATESTART > RATES.RATESTOP;
SPAFACTOR.SPACODE > SPAFACTOR.CODEDEF;
SPAFACTOR.SPACODE > SPAFACTOR.SPAFACTR;
SPAFACTOR.SPACODE > SPAFACTOR.SPADOLLARS;
TIMSHEET.SSN, TIMSHEET.ENDDATE > TIMSHEET.DEPT;
TIMSHEET.SSN, TIMSHEET.ENDDATE > TIMSHEET.MAINMO;
TIMSHEET.SSN, TIMSHEET.ENDDATE > TIMSHEET.OTTOTAL;
TIMSHEET.SSN, TIMSHEET.ENDDATE > TIMSHEET.TPNW;
TIMSHEET.SSN, TIMSHEET.ENDDATE > TIMSHEET.SPA;
TIMSHEET.SSN, TIMSHEET.ENDDATE > TIMSHEET.SEADAYS;
TIMDETAIL.SSN, TIMDETAIL.ENDDATE, TIMDETAIL.MORNO >
TIMDETAIL.STDHIRS;
TIMDETAIL.SSN, TIMDETAIL.ENDDATE, TIMDETAIL.MORNO >
TIMDETAIL.OTHRs;
TIMDETAIL.SSN, TIMDETAIL.ENDDATE, TIMDETAIL.MORNO >
TIMDETAIL.CHRGDEPT;
TIMDETAIL.SSN, TIMDETAIL.ENDDATE, TIMDETAIL.MORNO >
TIMDETAIL.SPA;
TIMDETAIL.SSN, TIMDETAIL.ENDDATE, TIMDETAIL.MORNO >
TIMDETAIL.SEADAYS;
CONTTIME.SSN, CONTTIME.TSTART > CONTTIME.TSTOP;
CONTTIME.SSN, CONTTIME.TSTART > CONTTIME.DEPT;
CONTTIME.SSN, CONTTIME.TSTART > CONTTIME.MAINMO;
CONTTIME.SSN, CONTTIME.TSTART > CONTTIME.STDHIRS;
CONTTIME.SSN, CONTTIME.TSTART > CONTTIME.OTHRs;
CONTTIME.SSN, CONTTIME.TSTART > CONTTIME.TPNW;
CONTTIME.SSN, CONTTIME.TSTART > CONTTIME.SPA;
CONTDetail.SSN, CONTDetail.TSTART, CONTDetail.MORNO >
CONTDetail.STDHIRS;
THIS IS THE LIST OF ATTRIBUTES WITH THEIR ABBREVIATIONS.

E00    EMPLOYEE.SSN
E01    EMPLOYEE.NAME
E02    EMPLOYEE.TITLE
E03    EMPLOYEE.DEPT
M00    MORDNOS.MORNO
M01    MORDNOS.MORTITLE
M02    MORDNOS.CONTTASK
M03    MORDNOS.CONT
M04    MORDNOS.MORTYPE
M05    MORDNOS.FND
D00    DEPTBUDG.DEPT
D01    DEPTBUDG.CONTTASK
D02    DEPTBUDG.DOLLARBUDG
D03    DEPTBUDG.STDHRBUDG
D04    DEPTBUDG.OTHRBUDG
R00    RATES.DEPT
R01    RATES.TITLE
R02    RATES.RATESTART
R03    RATES.DOMHR
R04    RATES.ROTDOT
R05    RATES.RATESTOP
S00    SPAFACTOR.SPACECODE
S01    SPAFACTOR.CODEDEF
S02    SPAFACTOR.SPAFACTR
S03    SPAFACTOR.SPADOLLARS
P11 PURCHORD, AMOUNT  
P12 PAYPERIOD, ENDDATE  
P13 PAYPERIOD, PERLENGTH

THE TOKENS MARKED *TRUE* ARE EXTRANEOUS IN THE FDS:
FD NUMBER : 001 TOKEN: E00
FD NUMBER : 002 TOKEN: E00
FD NUMBER : 003 TOKEN: E00
FD NUMBER : 004 TOKEN: M00
FD NUMBER : 005 TOKEN: M00
FD NUMBER : 006 TOKEN: M00
FD NUMBER : 007 TOKEN: M00
FD NUMBER : 008 TOKEN: M00
FD NUMBER : 009 TOKEN: D01
FD NUMBER : 009 TOKEN: D00
FD NUMBER : 010 TOKEN: D01
FD NUMBER : 010 TOKEN: D00
FD NUMBER : 011 TOKEN: D01
FD NUMBER : 011 TOKEN: D00
FD NUMBER : 012 TOKEN: R02
FD NUMBER : 012 TOKEN: R01
FD NUMBER : 012 TOKEN: R00
FD NUMBER : 013 TOKEN: R02
FD NUMBER : 013 TOKEN: R01
FD NUMBER : 013 TOKEN: R00
FD NUMBER : 014 TOKEN: R02
FD NUMBER : 014 TOKEN: R01

- 61 -
FD NUMBER : 014 TOKEN: R00
FD NUMBER : 015 TOKEN: S00
FD NUMBER : 016 TOKEN: S00
FD NUMBER : 017 TOKEN: S00
FD NUMBER : 018 TOKEN: T01
FD NUMBER : 018 TOKEN: T00
FD NUMBER : 019 TOKEN: T01
FD NUMBER : 019 TOKEN: T00
FD NUMBER : 020 TOKEN: T01
FD NUMBER : 020 TOKEN: T00
FD NUMBER : 021 TOKEN: T01
FD NUMBER : 021 TOKEN: T00
FD NUMBER : 022 TOKEN: T01
FD NUMBER : 022 TOKEN: T00
FD NUMBER : 023 TOKEN: T01
FD NUMBER : 023 TOKEN: T00
FD NUMBER : 024 TOKEN: T10
FD NUMBER : 024 TOKEN: T09
FD NUMBER : 024 TOKEN: T08
FD NUMBER : 025 TOKEN: T10
FD NUMBER : 025 TOKEN: T09
FD NUMBER : 025 TOKEN: T08
FD NUMBER : 026 TOKEN: T10
FD NUMBER : 026 TOKEN: T09
FD NUMBER : 026 TOKEN: T08
FD NUMBER : 027 TOKEN: T10
FD NUMBER : 027 TOKEN: T09
FD NUMBER : 027 TOKEN: T08
FD NUMBER : 028 TOKEN: T10
FD NUMBER : 028 TOKEN: T09
FD NUMBER : 028 TOKEN: T08
FD NUMBER : 029 TOKEN: C01
FD NUMBER : 029 TOKEN: C00
FD NUMBER : 030 TOKEN: C01
FD NUMBER : 030 TOKEN: C00
FD NUMBER : 031 TOKEN: C01
FD NUMBER : 031 TOKEN: C00
FD NUMBER : 032 TOKEN: C01
FD NUMBER : 032 TOKEN: C00
FD NUMBER : 033 TOKEN: C01
FD NUMBER : 033 TOKEN: C00
FD NUMBER : 034 TOKEN: C01
FD NUMBER : 034 TOKEN: C00
FD NUMBER : 035 TOKEN: C01
FD NUMBER : 035 TOKEN: C00
FD NUMBER : 036 TOKEN: C11
FD NUMBER : 036 TOKEN: C10
FD NUMBER : 036 TOKEN: C09
FD NUMBER : 037 TOKEN: C11
FD NUMBER: 037 TOKEN: C10
FD NUMBER: 037 TOKEN: C09
FD NUMBER: 038 TOKEN: C11
FD NUMBER: 038 TOKEN: C10
FD NUMBER: 038 TOKEN: C09
FD NUMBER: 039 TOKEN: C11
FD NUMBER: 039 TOKEN: C10
FD NUMBER: 039 TOKEN: C09
FD NUMBER: 040 TOKEN: V01
FD NUMBER: 040 TOKEN: V00
FD NUMBER: 041 TOKEN: V01
FD NUMBER: 041 TOKEN: V00
FD NUMBER: 042 TOKEN: V01
FD NUMBER: 042 TOKEN: V00
FD NUMBER: 043 TOKEN: V01
FD NUMBER: 043 TOKEN: V00
FD NUMBER: 044 TOKEN: P01
FD NUMBER: 044 TOKEN: P00
FD NUMBER: 045 TOKEN: P01
FD NUMBER: 045 TOKEN: P00
FD NUMBER: 046 TOKEN: P01
FD NUMBER: 046 TOKEN: P00
FD NUMBER: 047 TOKEN: P01
FD NUMBER: 047 TOKEN: P00
FD NUMBER: 048 TOKEN: P06
THE REDUNDANT FDS ARE MARKED *TRUE* :
<table>
<thead>
<tr>
<th>NUMBR020</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMBR021</td>
<td></td>
</tr>
<tr>
<td>NUMBR022</td>
<td></td>
</tr>
<tr>
<td>NUMBR023</td>
<td></td>
</tr>
<tr>
<td>NUMBR024</td>
<td></td>
</tr>
<tr>
<td>NUMBR025</td>
<td></td>
</tr>
<tr>
<td>NUMBR026</td>
<td></td>
</tr>
<tr>
<td>NUMBR027</td>
<td></td>
</tr>
<tr>
<td>NUMBR028</td>
<td></td>
</tr>
<tr>
<td>NUMBR029</td>
<td></td>
</tr>
<tr>
<td>NUMBR030</td>
<td></td>
</tr>
<tr>
<td>NUMBR031</td>
<td></td>
</tr>
<tr>
<td>NUMBR032</td>
<td></td>
</tr>
<tr>
<td>NUMBR033</td>
<td></td>
</tr>
<tr>
<td>NUMBR034</td>
<td></td>
</tr>
<tr>
<td>NUMBR035</td>
<td></td>
</tr>
<tr>
<td>NUMBR036</td>
<td></td>
</tr>
<tr>
<td>NUMBR037</td>
<td></td>
</tr>
<tr>
<td>NUMBR038</td>
<td></td>
</tr>
<tr>
<td>NUMBR039</td>
<td></td>
</tr>
<tr>
<td>NUMBR040</td>
<td></td>
</tr>
<tr>
<td>NUMBR041</td>
<td></td>
</tr>
<tr>
<td>NUMBR042</td>
<td></td>
</tr>
<tr>
<td>NUMBR043</td>
<td></td>
</tr>
</tbody>
</table>
FD NUMBR044
FD NUMBR045
FD NUMBR046
FD NUMBR047
FD NUMBR048
FD NUMBR049
FD NUMBR050
FD NUMBR051
FD NUMBR052
FD NUMBR053

THE FOLLOWING FDs HAVE THE SAME LHS AND ARE THEREFORE GROUPED TOGETHER INTO PARTITION CLASSES:

PARTITION CLASS NUMBER 001:053
PARTITION CLASS NUMBER 002:048049050051052
PARTITION CLASS NUMBER 003:047046045044
PARTITION CLASS NUMBER 004:043042041040
PARTITION CLASS NUMBER 005:036037038039
PARTITION CLASS NUMBER 006:03503403303203103029
PARTITION CLASS NUMBER 007:024025026027028
PARTITION CLASS NUMBER 008:023022021020019018
PARTITION CLASS NUMBER 009:015016017
PARTITION CLASS NUMBER 010:012013014
PARTITION CLASS NUMBER 011:011010009
PARTITION CLASS NUMBER 012:004005006007008
PARTITION CLASS NUMBER 013:001002003

001 002 003 004 005 006
007 008 009 010 011 012
013

THE FOLLOWING FDs ARE REDUNDANT AFTER ADDING THE BIJECTIONS TO THE FD STRUCTURE:

NONE
THIS IS THE SCHEMA IN 3NF:

(PAYPERIOD.ENDDATE) \rightarrow PAYPERIOD.PERLENGTH

(PURCHORD.PURCHORDNO) \rightarrow PURCHORD.REQPURCH
PURCHORD.ORDERDATE PURCHORD.MORNO PURCHORD.DEPT
PURCHORD.AMOUNT

(PETTYCASH.EMPEQNO PETTYCASH.SSN) \rightarrow PETTYCASH.CHRGDEPT
PETTYCASH.AMOUNT PETTYCASH.GN250DATE PETTYCASH.MORNO

(VOUCHER.VOUCHNO VOUCHER.SSN) \rightarrow VOUCHER.CHRGDEPT
VOUCHER.VOUCHDATE VOUCHER.AMOUNT VOUCHER.MORNO

(CONTDETAIL.MORNO CONTDetail.TSTART CONTDetail.SSN) \rightarrow
CONTDETAIL.STDHRS CONTDetail.OTHRS CONTDetail.CHRGDEPT
CONTDETAIL.SPA

(CONTTIME.TSTART CONTIME.SSN) \rightarrow CONTIME.SPA
CONTIME.TPNW CONTIME.OTHRS CONTIME.STDHRS
CONTIME.MAINMO CONTIME.DEPT CONTIME.TSTOP

(TIMDETAIL.MORNO TIMDETAIL.ENDDATE TIMDETAIL.SSN) \rightarrow
TIMDETAIL.STDHRS TIMDETAIL.OTHRS TIMDETAIL.CHRGDEPT
TIMDETAIL.SPA TIMDETAIL.SEADAYS

(TIMSHEET.ENDDATE TIMSHEET.SSN) \rightarrow TIMSHEET.SEADAYS
TIMSHEET.SPA TIMSHEET.TPNW TIMSHEET.OTTOTAL
TIMSHEET.MAINMO TIMSHEET.DEPT

(SPAFACTOR.SPACODE) \rightarrow SPAFACTOR.CODEDEF
SPAFACTOR.SPACFACTR SPAFACTOR.SPADOLLARS

(RATES.RATESTART RATES.TITLE RATES.DEPT) \rightarrow RATES.DOMHR
RATES.ROTDOT RATES.RATESTOP

(DEPTBUDG.CONTTASK DEPTBUDG.DEPT) \rightarrow DEPTBUDG.OTHRBUDG
DEPTBUDG.STDHRBUDG DEPTBUDG.DOLLARBUDG

- 68 -
(MORDNOS.MORNO ) > MORDNOS.MORTITLE MORDNOS.CONTTASK
MORDNOS.CONT  MORDNOS.MORTYPE  MORDNOS.FND

(EMPLOYEE.SSN  ) > EMPLOYEE.NAME  EMPLOYEE.TITLE
EMPLOYEE.DEPT
### EMPLOYEE TABLE

<table>
<thead>
<tr>
<th>Column name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>char(25)</td>
</tr>
<tr>
<td>ssn</td>
<td>char(11)</td>
</tr>
<tr>
<td>title</td>
<td>char(6)</td>
</tr>
<tr>
<td>dept</td>
<td>char(4)</td>
</tr>
</tbody>
</table>

Primary key = ssn

### MORD_NOS TABLE

<table>
<thead>
<tr>
<th>Column name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>mor_no</td>
<td>char(6)</td>
</tr>
<tr>
<td>mor_title</td>
<td>char(60)</td>
</tr>
<tr>
<td>cont_task</td>
<td>char(8)</td>
</tr>
<tr>
<td>cont</td>
<td>char(4)</td>
</tr>
<tr>
<td>mor_type</td>
<td>char(6)</td>
</tr>
<tr>
<td>fnd</td>
<td>char(3)</td>
</tr>
</tbody>
</table>

Primary key = mor_no

Figure 5. Global Schema Employee & Mord_nos
DEPT_BUDG TABLE

<table>
<thead>
<tr>
<th>Column name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>cont_task</td>
<td>char(8)</td>
</tr>
<tr>
<td>dept</td>
<td>char(4)</td>
</tr>
<tr>
<td>dollar_bdg</td>
<td>money(10,2)</td>
</tr>
<tr>
<td>stdhr_bdg</td>
<td>integer</td>
</tr>
<tr>
<td>othr_bdg</td>
<td>integer</td>
</tr>
</tbody>
</table>

Primary key = cont_task
depth

RATES TABLE

<table>
<thead>
<tr>
<th>Column name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>dept</td>
<td>char(4)</td>
</tr>
<tr>
<td>title</td>
<td>char(6)</td>
</tr>
<tr>
<td>dom_hr</td>
<td>money(5,2)</td>
</tr>
<tr>
<td>rotdot</td>
<td>money(5,2)</td>
</tr>
<tr>
<td>rate_start</td>
<td>date</td>
</tr>
<tr>
<td>rate_stop</td>
<td>date</td>
</tr>
</tbody>
</table>

Primary key = dept
title
rate_start

SPAFACTOR TABLE

<table>
<thead>
<tr>
<th>Column name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>spa_code</td>
<td>smallint</td>
</tr>
<tr>
<td>code_def</td>
<td>char(40)</td>
</tr>
<tr>
<td>spa_factr</td>
<td>decimal(4,3)</td>
</tr>
<tr>
<td>spa_dollars</td>
<td>money(5,2)</td>
</tr>
</tbody>
</table>

Figure 6. Global Schema Dept_budg, Rates, & Spafactor
TIMSHEET TABLE

<table>
<thead>
<tr>
<th>Column name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssn</td>
<td>char(11)</td>
</tr>
<tr>
<td>enddate</td>
<td>date</td>
</tr>
<tr>
<td>dept</td>
<td>char(4)</td>
</tr>
<tr>
<td>main_mo</td>
<td>char(6)</td>
</tr>
<tr>
<td>ot_total</td>
<td>smallint</td>
</tr>
<tr>
<td>tpnw</td>
<td>smallint</td>
</tr>
<tr>
<td>spa</td>
<td>smallint</td>
</tr>
<tr>
<td>sea_days</td>
<td>smallint</td>
</tr>
</tbody>
</table>

Primary key = ssn  
enddate

TIM_DETAIL TABLE

<table>
<thead>
<tr>
<th>Column name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssn</td>
<td>char(11)</td>
</tr>
<tr>
<td>enddate</td>
<td>date</td>
</tr>
<tr>
<td>mor_no</td>
<td>char(6)</td>
</tr>
<tr>
<td>stdhrs</td>
<td>smallint</td>
</tr>
<tr>
<td>othrs</td>
<td>smallint</td>
</tr>
<tr>
<td>chrgdept</td>
<td>char(4)</td>
</tr>
<tr>
<td>spa</td>
<td>smallint</td>
</tr>
<tr>
<td>sea_days</td>
<td>smallint</td>
</tr>
</tbody>
</table>

Primary key = ssn  
enddate  
mor_no

PAYPERIOD TABLE

<table>
<thead>
<tr>
<th>Column name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>per_length</td>
<td>smallint</td>
</tr>
<tr>
<td>enddate</td>
<td>date</td>
</tr>
</tbody>
</table>

Primary key = enddate

Figure 7. Global Schema Timsheet, Tim_detail, & Payperiod
### CONT_TIME TABLE

<table>
<thead>
<tr>
<th>Column name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssn</td>
<td>char(11)</td>
</tr>
<tr>
<td>tstart</td>
<td>date</td>
</tr>
<tr>
<td>tstop</td>
<td>date</td>
</tr>
<tr>
<td>dept</td>
<td>char(4)</td>
</tr>
<tr>
<td>main_mo</td>
<td>char(6)</td>
</tr>
<tr>
<td>std_total</td>
<td>smallint</td>
</tr>
<tr>
<td>ot_total</td>
<td>smallint</td>
</tr>
<tr>
<td>tpnw</td>
<td>smallint</td>
</tr>
<tr>
<td>spa</td>
<td>smallint</td>
</tr>
</tbody>
</table>

Primary key = ssn
tstart

### CONT_DETAIL TABLE

<table>
<thead>
<tr>
<th>Column name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssn</td>
<td>char(11)</td>
</tr>
<tr>
<td>tstart</td>
<td>date</td>
</tr>
<tr>
<td>mor_no</td>
<td>char(6)</td>
</tr>
<tr>
<td>stdhrs</td>
<td>smallint</td>
</tr>
<tr>
<td>othrs</td>
<td>smallint</td>
</tr>
<tr>
<td>chrgdept</td>
<td>char(4)</td>
</tr>
<tr>
<td>spa</td>
<td>smallint</td>
</tr>
</tbody>
</table>

Primary key = ssn
tstart
mor_no

Figure 8. Global Schema Cont_time & Cont_detail
**VOUCHER TABLE**

<table>
<thead>
<tr>
<th>Column name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssn</td>
<td>char(11)</td>
</tr>
<tr>
<td>vouch_date</td>
<td>date</td>
</tr>
<tr>
<td>mor_no</td>
<td>char(6)</td>
</tr>
<tr>
<td>amount</td>
<td>money(8,2)</td>
</tr>
<tr>
<td>vouch_no</td>
<td>char(9)</td>
</tr>
<tr>
<td>chrgdept</td>
<td>char(4)</td>
</tr>
</tbody>
</table>

Primary key = ssn  
vouch_no

**PURCH_ORD TABLE**

<table>
<thead>
<tr>
<th>Column name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>purch_ord_no</td>
<td>char(12)</td>
</tr>
<tr>
<td>req_purch</td>
<td>char(1)</td>
</tr>
<tr>
<td>order_date</td>
<td>date</td>
</tr>
<tr>
<td>mor_no</td>
<td>char(6)</td>
</tr>
<tr>
<td>dept</td>
<td>char(4)</td>
</tr>
<tr>
<td>amount</td>
<td>money(9,2)</td>
</tr>
</tbody>
</table>

Primary key = purch_ord_no

**Figure 9.** Global Schema Voucher & Purch_ord
### PETTYCASH TABLE

<table>
<thead>
<tr>
<th>Column name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssn</td>
<td>char(11)</td>
</tr>
<tr>
<td>gn250_date</td>
<td>date</td>
</tr>
<tr>
<td>mor_no</td>
<td>char(6)</td>
</tr>
<tr>
<td>amount</td>
<td>money(5,2)</td>
</tr>
<tr>
<td>chrgdept</td>
<td>char(4)</td>
</tr>
<tr>
<td>emp_seq_no</td>
<td>char(4)</td>
</tr>
</tbody>
</table>

[Primary key = ssn, emp_seq_no]

### GN300 TABLE

<table>
<thead>
<tr>
<th>Column name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>mor_no</td>
<td>char(6)</td>
</tr>
<tr>
<td>gn300_date</td>
<td>date</td>
</tr>
<tr>
<td>amount</td>
<td>money(7,2)</td>
</tr>
<tr>
<td>chrgdept</td>
<td>char(4)</td>
</tr>
</tbody>
</table>

**Figure 10. Global Schema Pettycash & GN300**
<table>
<thead>
<tr>
<th>Column name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>mor_no</td>
<td>char(6)</td>
</tr>
<tr>
<td>dept</td>
<td>char(4)</td>
</tr>
<tr>
<td>cont_task</td>
<td>char(8)</td>
</tr>
<tr>
<td>cont</td>
<td>char(4)</td>
</tr>
<tr>
<td>fnd</td>
<td>char(3)</td>
</tr>
<tr>
<td>lab</td>
<td>money(10,2)</td>
</tr>
<tr>
<td>clab</td>
<td>money(8,2)</td>
</tr>
<tr>
<td>mps</td>
<td>money(9,2)</td>
</tr>
<tr>
<td>vouch</td>
<td>money(8,2)</td>
</tr>
<tr>
<td>cvouch</td>
<td>money(8,2)</td>
</tr>
<tr>
<td>labstdhr</td>
<td>integer</td>
</tr>
<tr>
<td>labothrs</td>
<td>smallint</td>
</tr>
<tr>
<td>clabstdhr</td>
<td>integer</td>
</tr>
<tr>
<td>clabothr</td>
<td>smallint</td>
</tr>
</tbody>
</table>

Primary key = mor_no, dept

**Figure 11. Global Schema Summary Table**
Version Control and 2 Phase Commit Algorithm

/* This algorithm will demonstrate the steps required by both the central node (coordinator) and the local node when an update procedure is called. For this example we will assume that we are updating the mord_nos table. This table is fully replicated at all nodes. */

Central Node

/* First obtain network address, database name, table, and table version number from the database. A "$" at the beginning of any line indicates that this is an SQL statement. A "$" in front of a variable indicates that this variable was passed in from the PERFORM screen. */

$ database results;
$ select full_rep from rep_thls
    where table = $mord_nos;
if (full_rep = yes) /* table is fully replicated at each node */
then {
    $ select addr, db_name, table, version
        from net_addr, version_tbl
        where version_tbl.table = $mord_nos
            and net_addr.local_nodid =
                version_tbl.local_nodid
        into list_of_node_info;
    /* The above query will return a tuple for each local node */
else /* The table is not fully replicated, we will be updating at just one local node. */
$ select addr, db_name, table, version
    from net_addr, version_tbl
    where net_addr.local_nodid = $asst_manager
        and version_tbl.local_nodid = $asst_manager
        and table = $mord_nos
    into list_of_node_info;
} /* The above query will return a single tuple for the local_nodid stored in $asst_manager. */

BEGIN WORK
write update information to transaction log
for (all tuple(s) in list_of_node_info)
call Procedure update_local_node(update_attrib
[old_value, new_value], network_address,
table_name, version_no., db_name,
key_attrib[value])

/* This procedure now makes remote procedure calls to
local nodes and passes all of the variables it
received minus network_address */
wait(reply)

*************** Local Node ****************

Procedure update_replicated_table(update_attrib
[old_value, new_value], table_name, version_no.,
db_name, key_attrib[value])
$ select db_name;
$ select version from version_tbl
where table = $mord_nos;
if (version != version_no.) /* local table version number
does not match master table
version number at central
node */
then /* version will have to be brought up to date */
{ return(abort_bad_ver xx)
  wait(command)
  command(global_abort)
  return(ack_abort)
  exit /* Central node will now start another
  procedure to update the table with the bad
  version number and will then restart the
  transaction. */
else /* versions matched */
BEGIN WORK
write update information to transaction log
$ update $mord_nos
  set update_attrib = new_value
  where key_attrib = value;
$ update version_tbl
  set version = version + 1
  where table = $mord_nos;
}

if (update was not successful)
then
{ return(abort)
  wait(command)
  command of(global_abort)
  return(ack_abort)
  exit
else /* update was successful */
write ready to transaction log
return(ready)
wait(command)
}

*************** Central Node ***************

case (reply of)
abort_bad_ver xx /* from any node */
write global_abort to transaction log
send command(abort)
wait(reply)
reply of(ack_abort) /* assuming no lost messages */
ROLLBACK WORK /* undoes transaction */
call Procedure version_update(network_address,
db_name, table_name,
master_version_no.,
local_version_no.)
/* Access transaction log at central node and locate missing transaction(s). */
read transaction log /* extract missing transactions */
/* Make remote procedure call to procedure bad_version (1 phase commit) at local node. Pass the missing transactions. */
wait(reply)
reply of(ack_transact) /* 1 phase commit was successful */
return(ack_ver_updated) /* to procedure update_local_node */
exit
write "Updating Old Version" /* to PERFORM screen */
wait(reply) /* from version_update procedure */
reply of(ack_ver_updated)
restart transaction ;;
abort /* from any node */
write global_abort to transaction log
send command(abort)
wait(reply)
reply of(ack_abort) /* assuming no lost messages */
ROLLBACK WORK /* undoes transaction */
/* If this was first attempt to update, restart transaction, else write "Update Aborted by Node(s) xxxx" to PERFORM screen. */
no reply /* timeout expired: from any node */
write global_abort to transaction log
send command(abort)
wait(reply)
reply of(ack_abort)
ROLLBACK WORK
/* If this was first attempt to update, restart
transaction, else write "Update Aborted by
Node(s) xxxx" to PERFORM screen. */

ready /* from all nodes */
$ update version_tbl
   set version = version + 1
   where table = $mord_nos;
   /* The above increments mord_nos' version
   number by 1 for each node including the
   central node's tuple */
write global_commit to transaction log
send command(commit)
wait(reply)
reply of(ack_commit)
COMMIT WORK
exit

************************ Local Node *********************

if (command = abort)
then
   { ROLLBACK WORK
     return(ack_abort)
     exit
   else /* command was commit */
     COMMIT WORK
     return(ack_commit)
   }
exit

END of Version control and 2 phase commit algorithm.
THE DESIGN
OF A
DISTRIBUTED DATABASE
AND
A REPLICATED DATA MANAGEMENT ALGORITHM

by

Steven J. Van Buren

B. S., Michigan Technological University, 1972

AN ABSTRACT OF A REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1988
A design for a distributed database titled the Financial Control System is described. The purpose of the Financial Control System is to track incurrences on Federal contracts managed by the Ocean Systems Organization of AT&T Technologies' Federal Systems Division. The distributed database is designed to be implemented using a commercial centralized database management system (DBMS) as a component of the distributed system. An algorithm designed to manage replicated data within the distributed database is presented. The algorithm integrates a version control scheme with a two phase commit protocol.