Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

60

M. J. Flynn, J. N. Gray, A. K. Jones, K. Lagally
H. Opderbeck, G. J. Popek, B. Randell
J. H. Saltzer, H. R. Wehle

Operating Systems

An Advanced Course

Edited by

R. Bayer, R. M. Graham, and G. Seegmüller

Springer-Verlag

Berlin Heidelberg New York 1978

Notes on Data Base Operating Systems

Jim Gray
IBM Research Laboratory
San Jose, California. 95193

Summer 1977

ACKNOWLEDGMENTS

This paper plagiarizes the work of the large and anonymous army of people working in the field. Because of the state of the field, there are few references to the literature (much of the “literature” is in internal memoranda, private correspondence, program logic manuals and prologues to the source language listings of various systems.)

The section on data management largely reflects the ideas of Don Chamberlin, Ted Codd, Chris Date, Dieter Gawlick, Andy Heller, Frank King, Franco Putzolu, and Bob Taylor. The discussion of views is abstracted from a paper co-authored with Don Chamberlin and Irv Traiger.

The section on data communications stems from conversations with Denny Anderson, Homer Leonard, and Charlie Sanders.

The section on transaction scheduling derives from discussions with Bob Jackson and Thomas Work.

The ideas on distributed transaction management are an amalgam of discussions with Honor Leonard.

Bruce Lindsay motivated the discussion of exception handling.

The presentation of consistency and locking derives from discussions and papers co-authored with Kapali Eswaran, Raymond Lorie, Franco Putzolu and Irving Traiger. Also Ron Obermark (IMS program isolation), Phil Macri (DMS 1100), and Paul Roever clarified many locking issues for me.

The presentation of recovery is co-authored with Paul McJones and John Nauman. Dieter Gawlick made many valuable suggestions. It reflects the ideas of Mike Blasgen, Dar Busa, Ron Obermark, Earl Jenner, Tom Price, Franco Putzolu, Butler Lampson, Howard Sturgis and Steve Weick.

All members of the System R group (IBM Research, San Jose) have contributed materially to this paper.

I am indebted to Mike Blasgen, Dieter Gawlick, especially to John Nauman, each of whom made many constructive suggestions about earlier drafts of these notes.

If you feel your ideas or work are inadequately plagiarized, please annotate this manuscript and return it to me.

1. INTRODUCTION

Most large institutions have now heavily invested in a data base system. In general they have automated such clerical tasks as inventory control, order entry, or billing. These systems often support a worldwide network of hundreds of terminals. Their purpose is to reliably store and retrieve large quantities of data. The life of many institutions is critically dependent on such systems, when the system is down the corporation has amnesia.

This puts an enormous burden on the implementers and operators of such systems. The systems must on the one hand be very high performance and on the other hand they must be very reliable.

1.1. A SAMPLE SYSTEM

Perhaps it is best to begin by giving an example of such a system. A large bank may have one thousand teller terminals (several have 20,000 tellers but at present no single system supports such a large network). For each teller, there is a record describing the teller's cash drawer and for each branch there is a record describing the cash position of that branch (bank general ledger), It is likely to have several million demand deposit accounts (say 10,000,000 accounts). Associated with each account is a master record giving the account owner, the account balance, and a list of recent deposits and withdrawals applicable to this account. This database occupies over 10,000,000,000 bytes and must all be on-line at all times,

The database is manipulated with application dependent transactions, which were written for this application when it was installed. There are many transactions defined on this database to query it and update it. A particular user is allowed to invoke a subset of these transactions. Invoking a transaction consists of typing a message and pushing a button. The teller terminal appends the transaction identity, teller identity and terminal identity to the message and transmits it to the central data manager. The data communication manager receives the message and translates it to some canonical form.

It then passes the message to the transaction manager, which validates the teller's authorization to invoke the specified transaction and then allocates and dispatches an instance of the transaction. The transaction processes the message, generates a response, and terminates. Data communications delivers the message to the teller.

Perhaps the most common transaction is in this environment is the DEBIT_CREDIT transaction which takes in a message from any teller, debits or credits the appropriate account (after running some validity checks), adjusts the teller cash drawer and branch balance, and then sends a response message to the teller. The transaction flow is:

```
     DEBIT_CREDIT:
          BEGIN_TRANSACTION;
          GET MESSAGE;
          EXTRACT ACCOUT_NUMBER, DELTA, TELLER, BRANCH
               FROM MESSAGE;
          FIND ACCOUNT(ACCOUT_NUMBER) IN DATA BASE;
          IF NOT_FOUND  | ACCOUNT_BALANCE + DELTA < 0 THEN
               PUT NEGATIVE RESPONSE;
          ELSE DO;
               ACCOUNT_BALANCE = ACCOUNT_BALANCE + DELTA;
               POST HISTORY RECORD ON ACCOUNT (DELTA);
               CASH_DRAWER(TELLER) = CASH_DRAWER(TELLER) + DELTA;
               BRANCH_BALANCE(BRANCH) = BRANCH_BALANCE(BRANCH) + DELTA;
               PUT MESSAGE ('NEW BALANCE =' ACCOUNT_BALANCE);
               END;
          COMMIT;
```

At peak periods the system runs about thirty transactions per second with a response time of two seconds.

The DEBIT_CREDIT transaction is very “small”. There is another class of transactions that behave rather differently. For example, once a month a transaction is run which produces a summary statement for each account. This transaction might be described by:

```
MONTHLY_STATEMENT:
     ANSWER :: =  SELECT *
                    FROM ACCOUNT, HISTORY
                   WHERE ACCOUNT. ACCOUNT_NUMBER = HISTORY. ACCOUNT_NUMBER
                   AND HISTORY_DATE  > LAST_REPORT
                   GROUPED BY ACCOUNT. ACCOUNT_NUMBER,
                   ASCENDING BY ACCOUNT. ACCOUNT_ADDRESS;
```

That is, collect all recent history records for each account and place them clustered with the account record into an answer file. The answers appear sorted by mailing address.

If each account has about fifteen transactions against it per month then this transaction will read 160,000,000 records and write a similar number of records. A naive implementation of this transaction will take 80 days to execute (50 milliseconds per disk seek implies two million seeks per day.) However, the system must run this transaction once a month and it must complete within a few hours.

There is a broad spread of transactions between these two types. Two particularly interesting types of transactions are conversational transactions that carry on a dialogue with the user and distributed transactions that access data or terminals at several nodes of a computer network,

Systems of 10,000 terminals or 100,000,000,000 bytes of on-line data or 150 transactions per second are generally considered to be the limit of present technology (software and hardware).

1.2. RELATIONSHIP TO OPERATING SYSTEM

If one tries to implement such an application on top of a general-purpose operating system it quickly becomes clear that many necessary functions are absent from the operating system. Historically, two approaches have been taken to this problem:

- Write a new, simpler and “vastly superior” operating system.
- Extend the basic operating system to have the desired function.

The first approach was very popular in the mid-sixties and is having a renaissance with the advent of minicomputers. The initial cost of a data management system is so low that almost any large customer can justify “rolling his own”. The performance of such tailored systems is often ten times better than one based on a general purpose system. One must trade this off against the problems of maintaining the system as it grows to meet new needs and applications. Group’s that followed this path now find themselves maintaining a rather large operating system, which must be modified to support new devices (faster disks, tape archives,...) and new protocols (e. g. networks and displays.) Gradually, these systems have grown to include all the functions of a general-purpose operating system. Perhaps the most successful approach to this has been to implement a hypervisor that runs both the data management operating system and some non-standard operating system. The "standard” operating system runs when the data manager is idle. The hypervisor is simply an interrupt handler which dispatches one or another system.

The second approach of extending the basic operating system is plagued with a different set of difficulties. The principal problem is the performance penalty of a general-purpose operating system. Very few systems are designed to deal with very large files, or with networks of thousands of nodes. To take a specific example, consider the process structure of a general-purpose system: The allocation and deallocation of a process should be very fast (500 instructions for the pair is expensive) because we want to do it 100 times per second. The storage occupied by the process descriptor should also be small (less than 1000 bytes.) Lastly, preemptive scheduling of processes makes no sense since they are not CPU bound (they do a lot of I/O). A typical system uses 16,000 bytes to represent a process and requires 200,000 instructions to allocate and deallocate this structure (systems without protection do it cheaper.) Another problem is that the general-purpose systems have been designed for batch and time-sharing operation. They have not paid sufficient attention to issues such as continuous operation: keeping the system up for weeks at a time and gracefully degrading in case of some hardware or software error.

1.3. GENERAL STRUCTURE OF DATA MANAGEMENT SYSTEMS

These notes try to discuss issues that are independent of which operating system strategy is adopted. No matter how the system is structured, there are certain problems it must solve. The general structure common to several data management systems is presented. Then two particular problems within the transaction management component are discussed in detail: concurrency control (locking) and system reliability (recovery).

This presentation decomposes the system into four major components:

- Dictionary: the central repository of the description and definition of all persistent system objects.
- Data Communications: manages teleprocessing lines and message traffic.
- Data Base manager: manages the information stored in the system.
- Transaction Management: manages system resources and system services such as locking and recovery.

Each of these components calls one another and in turn depends on the basic operating system for services.

1.3. BIBLIOGRAPHY

These notes are rather nitty-gritty; they are aimed at system implementers rather than at users. If this is the wrong level of detail for you (is too detailed) then you may prefer the very readable books:

Martin, *Computer Data Base Organization*, Prentice Hall, 1977 (What every DP Vice President should know.)

Martin, *Computer Data Base Organization, (2nd edition),* Prentice Hall, 1976 (What every application programmer should know.)

The following is a brief list of sane of the more popular general-purpose data management systems that are commercially available:

Airlines Control Program, International Business Machines Corporation

Customer Information Computer System, International Business Machines Corporation

Data Management System 1100, Sperry Univac Corporation

Extended Data Management system, Xerox Corporation

Information Management System /Virtual Systems, International Business Machines Corporation

Integrated Database Management System, Cullinane Corporation

Integrated Data Store/1, Honeywell Information Systems Inc

Model 204 Data Base Management System, Computer Corporation of America

System 2000, MRI Systems Corporation.

Total, Cincom Systems Corporation

Each of these manufacturers will be pleased to provide you with extensive descriptions of their systems.

Several experimental systems are under construction at present. Some of the more interesting are:

Astrahan et. al., “System R: a Relational Approach to Database Management”, Astrahan et. al., ACM Transactions on Database Systems, Vol. 1, No. 2, June 1976.

Marill and Stern, "The Datacomputer-A Network Data Utility.” Proc. 1975 National Computer Conference, AFIPS Press, 1975,

Stonebraker et. al., "The Design and Implementation of INGRESS.” ACM Transactions on Database Systems, Vol. 1, No. 3, Sept 1976,

There are very few publicly available case studies of data base usage. The following are interesting but may not be representative:

IBM Systems Journal, Vol. 16, No. 2, June 1977. (Describes the facilities and use of IMS and ACP).

“IMS/VS Primer," IBM World Trade Systems Center, Palo Alto California, Form number S320-5767-1, January 1977.

"Share Guide IMS User Profile, A Summary of Message Processing Program Activity in Online IMS Systems" IBM Palo Alto-Raleigh Systems Center Bulletin, form number 6320-6005, January 1977

Also there is one “standard” (actually “proposed standard" system):

CODASYL Data Base Task Group Report, April 1971. Available from ACM

2. DICTIONARY

2.1. WHAT IT IS

The description of the system, the databases, the transactions, the telecommunications network, and of the users are all collected in the dictionary. This repository:

- Defines the attributes of objects such as databases and terminals.
- Cross-references these objects.
- Records natural language (e. g. German) descriptions of the meaning and use of objects.

When the system arrives, the dictionary contains only a very few definitions of transactions (usually utilities), defines a few distinguished users (operator, data base administrator,...), and defines a few special terminals (master console). The system administrator proceeds to define new terminals, transactions, users, and databases. (The system administrator function includes data base administration (DBA) and data communications (network) administration (DCA). Also, the system administrator may modify existing definitions to match the actual system or to reflect changes. This addition and modification process is treated as an editing operation.

For example, one defines a new user by entering the “define” transaction and selecting USER from the menu of definable types. This causes a form to be displayed, which has a field for each attribute of a user. The definer fills in this form and submits it to the dictionary. If the form is incorrectly filled out, it is redisplayed and the definer corrects it. Redefinition follows a similar pattern; the current form is displayed, edited and then submitted. (There is also a non-interactive interface to the dictionary for programs rather than people.)

All changes are validated by the dictionary for syntactic and semantic correctness. The ability to establish the correctness of a definition is similar to ability of a compiler to detect the correctness of a program. That is, many semantic errors go undetected. These errors are a significant problem.

Aside from validating and storing definitions, the dictionary provides a query facility which answers questions such as: "Which transactions use record type A of file B?" or, "What are the attributes of terminal 34261".

The dictionary performs one further service, that of compiling the definitions into a "machine readable" form more directly usable by the other system components. For example, a terminal definition is converted from a variable length character string to a fixed format “descriptor” giving the terminal attributes in non-symbolic form.

The dictionary is a database along with a set of transactions to manipulate this database. Some systems integrate the dictionary with the data management system so that the data definition and data manipulation interface are homogeneous. This has the virtue of sharing large bodies of code and of providing a uniform interface to the user. Ingress and System R are examples of such systems.

Historically, the argument against using the database for the dictionary has been performance. There is very high read traffic on the dictionary during the normal operation of the system. A user logon requires examining the definitions of the user, his terminal, his category, and of the session that his logon establishes. The invocation of a transaction requires examining his authorization, the transaction, and the transaction descriptor (to build the transaction.) In turn the transaction definition may reference databases and queues which may in turn reference files, records and fields. The performance of these accesses is critical because they appear in the processing of each transaction. These performance constraints combined with the fact that the accesses are predominantly read-only have caused most systems to special-case the dictionary. The dictionary definitions and their compiled descriptors are stored by the data base management component. The dictionary-compiled descriptors are stored on a special device and a cache of them is maintained in high-speed storage on an LRU (Least Recently Used) basis. This mechanism generally uses a coarse granularity of locks and because operations are read only it keeps no log. Updates to the descriptors are made periodically while the system is quiesced.

The descriptors in the dictionary are persistent. During operation, many other short-lived descriptors are created for short-lived objects such as cursors, processes, and messages. Many of these descriptors are also kept in the descriptor cache.

The dictionary is the natural extension of the catalog or file system present in operating systems. The dictionary simply attaches more semantics to the objects it stores and more powerful operators on these objects.

Readers familiar with the literature may find a striking similarity between the dictionary and the notion of conceptual schema, which is “a model of the enterprise”. The dictionary is the conceptual schema without its artificial intelligence aspects. In time the dictionary component will evolve in the direction suggested by papers on the conceptual schema.

2.2. BIBLIOGRAPHY

DB/DC Data Dictionary General Information Manual, IBM, Form number GH20-9104-1, May 1977

UCC TEN, Technical Information Manual, University Computing, Corporation, 1976

Lefkovits, *Data Dictionary Systems*, Q. E. D. Information Sciences Inc., 1977, (A buyer's guide for data dictionaries.)

Nijssen (editor), *Modeling in Data Base Management Systems*, North Holland, 1976. (All you ever wanted about conceptual schema.)

"SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control." Chamberlain et. al., IBM Journal of Research and Development, Vol. 20, No. 6, November 1976. (Presents a unified data definition, data manipulation facility.)

3. DATA MANAGEMENT

The Data management component stores and retrieves sets of records. It implements the objects: network, set of records, cursor, record, field, and view.

3.1. RECORDS AND FIELDS

A record type is a sequence of field types, and a record instance is a corresponding sequence of field instances. Record types and instances are persistent objects. Record instances are the atomic units of insertion and retrieval. Fields are sub-objects of records and are the atomic units of update. Fields have the attributes of atoms (e. g. FIXED(31)or CHAR(*)) and field instances have atomic values (e. g. "3" or "BUTTERFLY"). Each record instance has a unique name called a record identifier (RID).

A field type constrains the type and values of instances of a field and defines the representation of such instances. The record type specifies what fields occur in instances of that record type.

A typical record might have ten fields and occupy 256 bytes although records often have hundreds of fields (e. g. a record giving statistics on a census tract has over 600 fields), and may be very large (several thousand bytes). A very simple record (nine fields and about eighty characters) might be described by:

```
DECLARE 1 PHONE_BOOK_RECORD,
     2 PERSON_NAME CHAR(*),
     2 ADDRESS,
          3 STREET_NUMBER CHAR(*),
          3 STREET_NAME CHAR(*),
          3 CITY CHAR(*),
          3 STATE CHAR(*),
          3 ZIP_CODE CHAR(5).
     2 PHONE_NUMBER,
          3 AREA_CODE CHAR(3),
          3 PREFIX CHAR(3),
          3 STATION CHAR(4);
```

The operators on records include INSERT, DELETE, FETCH, and UPDATE. Records can be CONNECTED to and DISCONNECTED from membership in a set (see below). These operators actually apply to cursors, which in turn point to records.

The notions of record and field correspond very closely to the notions of record and element in COBOL or structure and field in PL/I. Records are variously called entities, segments, tuples, and rows by different subcultures. Most systems have similar notions of records although they may or may not support variable length fields, optional fields (nulls), or repeated fields.

3.2. SETS

A set is a collection of records. This collection is represented by and implemented as an "access path" that runs through the collection of records. Sets perform the functions of :

- Relating the records of the set.
- In some instances directing the physical clustering of records in physical storage.

A record instance may occur in many different sets but it may occur at most once in a particular set.

There are three set types of interest:

- **Sequential set**: the records in the set form a single sequence. The records in the set are ordered either by order of arrival (entry sequenced (ES)), by cursor position at insert (CS), or are ordered (ascending or descending) by some subset of field values (key sequenced (KS)). Sequential sets model indexed-sequential files (ISAM, VSAM).
- **Partitioned set:** The records in the set form a sequence of disjoint groups of sequential sets. Cursor operators allow one to point at a particular group. Thereafter the sequential set operators are used to navigate within the group. The set is thus major ordered by hash and minor ordered (ES, CS or KS) within a group. Hashed files in which each group forms a hash bucket are modeled by partitioned sets,
- **Parent-child set**: The records of the set are organized into a two-level hierarchy. Each record instance is either a parent or a child (but not both). Each child has a unique parent and no children. Each parent has a (possibly null) list of children. Using parent-child sets one can build networks and hierarchies. Positional operators on parent-child sets include the operators to locate parents, as well as operations to navigate on the sequential set of children of a parent. The CONNECT and DISCONNECT operators explicitly relate a child to a parent, One obtains implicit connect and disconnect by asserting that records inserted in one set should also be connected to another. (Similar rules apply for connect, delete and update.) Parent-child sets can be used to support hierarchical and network data models.

A partitioned set is a degenerate form of a parent-child set (the partitions have no parents), and a sequential set is a degenerate form of a partitioned set (there is only one partition.) In this discussion care has been taken to define the operators so that they also subset. This has the consequence that if the program uses the simplest model it will be able to run on any data and also allows for subset implementations on small computers.

Inserting a record in one set map trigger its connection to several other sets. If set “I” is an index for set “F” then an insert, delete and update of a record in “F” may trigger a corresponding insert, delete, or update in set “I”. In order to support this, data manager must know:

- That insertion, update or deletion of a record causes its connection to, movement in, or disconnection from other sets.
- Where to insert the new record in the new set:
  - For sequential sets, the ordering must be either key sequenced or entry sequenced.
  - For partitioned sets, data manager must know the partitioning rule and know that the partitions are entry sequenced or key sequenced.
  - For parent-child sets, the data manager must know that certain record types are parents and that others are children. Further, in the case of children, data manager must be able to deduce the parent of the child.

We will often use the term “file” as a synonym for set.

3.3. CURSORS.

A cursor is "opened" on a specific set and thereafter points exclusively to records in that set. After a cursor is opened it may be moved, copied, or closed. While a cursor is opened it may be used to manipulate the record it addresses.

Records are addressed by cursors. Cursors serve the functions of:

- Pointing at a record.
- Enumerating all records in a set.
- Translating between the stored record format and the format visible to the cursor user. A simple instance of this might be a cursor that hides some fields of a record. This aspect will be discussed with the notion of view.

A cursor is an ephemeral object that is created from a descriptor when a transaction is initiated or during transaction execution by an explicit OPEN_CURSOR command. Also one may COPY_CURSOR a cursor to make another instance of the cursor with independent positioning. A cursor is opened on a specific set (which thereby defines the enumeration order (next) of the cursor.) A cursor is destroyed by the CLOSE_CURSOR command.

3.3.2. OPERATIONS ON CURSORS

Operators on cursors include:

```
FETCH (<cursor> [, <position>]) [HOLD] RETURNS(<record>)
```

Which retrieves the record pointed at by the named cursor. The record is moved to the specified target. If the position is specified the cursor is first positioned. If HOLD is specified the record is locked for update (exclusive), otherwise the record is locked in share mode.

```
INSERT (<cursor>[, <position>], <record>)
```

Inserts the specified record into the set specified by cursor. If the set is key sequenced or entry sequenced then the cursor is moved to the correct position before the record is inserted, otherwise the record is inserted at (after) the current position of the cursor in the set. If the record type automatically appears in other sets, it also inserted in them.

```
UPDATE (<cursor> [, <position>],<new_record>)
```

If position is specified the cursor is first positioned. The new record is then inserted in the set at the cursor position replacing the record pointed at by the cursor. If the set is sequenced by the updated fields, this may cause the record and cursor to move in the set.

```
DELETE (<cursor> [, <position>])
```

Deletes the record pointed at by the cursor after optionally repositioning the cursor.

```
MOVE_CURSOR (<cursor>, <position>) HOLD
```

Repositions the cursor in the set.

3.3.3 CURSOR POSITIONING

A cursor is opened to traverse a particular set. Positioning expressions have the syntax:

```
--+-------------<RID>-------------------+-------------+-;
  +-------------FIRST-------------------+             |
  +--------------N-TH-------------------+   +-CHILD---+
  +--------------LAST ------------------+---+---------+
  +---NEXT-----+                        +   +-PARENT--+
  +--PREVIOUS--+-<SELECTION EXPRESSION>-+   +-GROUP---+
  +------------+
```

where RID, FIRST , N-th, and LAST specify specific record occurrences while the other options specify the address relative to the current cursor position. It is also possible to set a cursor from another cursor.

The selection expression may be any Boolean expression valid for all record types in the set. The selection expression includes the relational operators: =, !=, >, <, <=, >=, and for character strings a "matches-prefix" operator sometimes called generic key. If next or previous is specified, the set must be searched sequentially because the current position is relevant. Otherwise, the search can employ hashing or indices to locate the record. The selection expression search may be performed via an index, which maps field values into RIDs.

Examples of commands are:

```
     FETCH   (CURSOR1, NEXT NAME='SMITH') HOLD RETURNS (POP);
     DELETE  (CURSOR1, NEXT NAME='JOE' CHILD);
     INSERT  (CURSOR1, , NEWCHILD);
```

For partitioned sets one may point the cursor at a specific partition by qualifying these operators by adding the modifier GROUP. A cursor on a parent-child (or partitioned) set points to both a parent record and a child record (or group and child within group). Cursors on such sets have two components: the parent or group cursor and the child cursor. Moving the parent curser, positions the child cursor to the first record in the group or under the parent. For parent-child sets one qualifies the position operator with the modifier NEXT_PARENT in order to locate the first child of the next parent or with the modifier WITHIN PARENT if the search is to be restricted to children of the current parent or group. Otherwise positional operators operate on children of the current parent.

There are rather obscure issues associated with cursor positioning.

The following is a good set of rules:

- A cursor can have the following positions:
  - Null.
  - Before the first record.
  - At a record.
  - Between two records.
  - After the last record.
- If the cursor points at a null set, then it is null. If the cursor points to a non-null set then it is always non-null.
- Initially the cursor is before the first record unless the OPEN_CURSOR specifies a position.
- An INSERT operation leaves the cursor pointing at the new record.
- A DELETE operation leaves the cursor between the two adjacent records , or at the top if there is no previous record, or at the bottom if there is a previous but no successor record.
- A UPDATE operation leaves the cursor pointing at the updated record.
- If an operation fails the cursor is not altered.

3.4. VARIOUS DATA MODELS

Data models differ in their notion of set.

3.4.1. RELATIONAL DATA MODEL

The relational model restricts itself to homogeneous (only one record type) sequential sets. The virtue of this approach is its simplicity and the ability to define operators that “distribute" over the set, applying uniformly to each record of the set. Since much of data processing involves repetitive operations on large volumes of data, this distributive property provides a concise language to express such algorithms. There is a strong analogy here with APL that uses the simple data structure of array and therefore is able to define powerful operators that work for all arrays. APL programs are very short and much of the control structure of the program is hidden inside of the operators.

To give an example of this, a “relational” program to find all overdue accounts in an invoice file might be:

```
SELECT ACCOUNT_NUMBER
FROM INVOICE
WHERE DUE_DATE<TODAY;
```

This should be compared to a PL/I program with a loop to get next record, and test for DUE_DATE and END_OF_FILE. The MONTHLY_STATEMENT transaction described in the introduction is another instance of the power and usefulness of relational operators.

On the other hand, if the work to be done does not involve processing many records, then the relational model seems to have little advantage over other models. Consider the DEBIT_CREDIT transaction which (1) reads a message from a terminal, (2) finds an account, (3) updates the account, (4) posts a history record, (5) updates the teller cash drawer, (6) updates the branch balance, and (7)puts a message to the terminal. Such a transaction would benefit little from relational operators (each operation touches only one record.)

One can define aggregate operators that distribute over hierarchies or networks. For example, the MAPLIST function of LISP distributes an arbitrary function over an arbitrary data structure.

3.4.2. HIERARCHICAL DATA MODEL

Hierarchical models use parent-child sets in a stylized way to produce a forest (collection of trees)of records. A typical application might use the three record types: LOCATIONS, ACCOUNTS, and INVOICES and two parent-child sets to construct the following hierarchy: All the accounts at a location are clustered together and all outstanding invoices of an account are clustered with the account, That is, a location has its accounts as children and an account has its invoices as children. This may be depicted schematically by:

```
+-----------+
| LOCATIONS |
+-----------+
      |
      |
+-----------+
| ACCOUNTS  |
+-----------+
      |
      |
+-----------+
| INVOICES  |
+-----------+
```

This structure has the advantage that records used together may appear clustered together in physical storage and that information common to all the children can be factored into the parent record. Also, one may quickly find the first record under a parent and deduce when the last has been seen without scanning the rest of the database.

Finding all invoices for an account received on a certain day involves positioning a cursor on the location, another cursor on the account number under that location, and a third cursor to scan over the invoices:

```
SET CURSOR1 to LOCATION=NAPA;
SET CURSOR2 TO ACCOUNT=FREEMARK_ABBEY;
SET CURSOR3 BEFORE FIRST_CHILD(CURSOR2);
DO WHILE  (! END_OF_CHILDREN):
     FETCH (CURSOR3) NEXT CHILD; DO_SOMETHING;
END;
```

Because this is such a common phenomenon, and because in a hierarchy there is only one path to a record, most hierarchical systems abbreviate the cursor setting operation to setting the lowest cursor in the hierarchy by specifying a "fully qualified key" or path from the root to the leaf (the other cursors are set implicitly.) In the above example:

```
SET CURSOR3 TO LOCATION=NAPA,
     ACCOUNT=FREEMARK_ABBEY,
     INVOICE_ANY;
DO WHILE (! END_OF_CHILDREN):
     FETCH (CURSOR3) NEXT CHILD; DO_SOMETHING;
END;
```

Which implicitly sets up cursors one and two.

The implicit record naming of the hierarchical model makes programming much simpler than for a general network. If the data can be structured as a hierarchy in some application then it is desirable to use this model to address it.

3.4.3. NETWORK DATA MODEL

Not all problems conveniently fit a hierarchical model. If nothing else, different users may want to see the same information in a different hierarchy. For example an application might want to see the hierarchy “upside-down” with invoice at the top and location at the bottom. Support for logical hierarchies (views) requires that the data management system support a general network. The efficient implementation of certain relational operators (sort-merge or join) also require parent-child sets and so require the full capability of the network data model.

The general statement is that if all relationships are nested one-to-many mappings then the data can be expressed as a hierarchy. If there are many-to-many mappings then a network is required. To consider a specific example of the need for networks, imagine that several locations may service the same account and that each location services several accounts, Then the hierarchy introduced in the previous section would require either that locations be subsidiary to accounts and be duplicated or that the accounts record be duplicated in the hierarchy under the two locations. This will give rise to complexities about the account having two balances..... A network model would allow one to construct the structure:

```
+----------+      +----------+
| LOCATION |      | LOCATION |
+----------+      +----------+
   |   |             |    |
   +---)-------------+    |
   |   |                  |
   |   +------------------+
   |                      |
   V                      V
+----------+      +----------+
| ACCOUNT  |      | ACCOUNT  |
+----------+      +----------+
   |                      |
   |                      |
   V                      V
+----------+      +----------+
| INVOICE  |      | INVOICE  |
+----------+      +----------+
```

A network built out of two parent-child sets.

3.4.4. COMPARISON OF DATA MODELS

By using “symbolic” pointers (keys), one may map any network data structure into a relational structure. In that sense all three models are equivalent, and the relational model is completely general. However, there are substantial differences in the style and convenience of the different models. Analysis of specific cases usually indicates that associative pointers (keys) cost three page faults to follow (for a multi-megabyte set) whereas following a direct pointer costs only one page fault. This performance difference explains why the equivalence of the three data models is irrelevant. If there is heavy traffic between sets then pointers must be used. (High-level languages can hide the use of these pointers.)

It is my bias that one should resort to the more elaborate model only when the simpler model leads to excessive complexity or to poor performance.

3.5. VIEWS

Records, sets, and networks that are actually stored are called base objects. Any query evaluates to a virtual set of records which may be displayed on the user's screen, fed to a further query, deleted from an existing set, inserted into an existing set, or copied to form a new base set. More importantly for this discussion, the query definition may be stored as a named m. The principal difference between a copy and a view is that updates to the original sets that produced the virtual set will be reflected in a view but will not affect a copy. A view is a dynamic picture of a query whereas a copy is a static picture.

There is a need for both views and copies. Someone wanting to record the monthly sales volume of each department might run the following transaction at the end of each month (an arbitrary syntax):

```
MONTHLY_VOLUME=
     SELECT DEPARTMENT, SUM(VOLUME)
     FROM SALES GROUPED BY DEPARTMENT;
```

The new base set MONTHLY_VOLUME is defined to hold the answer. On the other hand, the current volume can be gotten by the view:

```
DEFINE CURRENT_VOLUME (DEPARTMENT, VOLUME) VIEW AS:
          SELECT DEPARTMENT, SUM(VOLUME)
          FROM SALES
          GROUPED BY DEPARTMENT;
```

Thereafter, any updates to the SALES set will be reflected in the CURRENT_VOLUME view. Again, CURRENT_VOLUME may be used in the same ways base sets can be used. For example one can compute the difference between the current and monthly volume.

The semantics of views are quite simple. Views can be supported by a process of substitution in the abstract syntax (parse tree) of the statement. Each time a view is mentioned, it is replaced by its definition.

To summarize, any query evaluates to a virtual set. Naming this virtual set makes it a view. Thereafter, this view can be used as a set. This allows views to be defined as field and record subsets of sets, statistical summaries of sets and more complex combinations of sets.

There are three major reasons for defining views:

- **Data independence**: giving programs a logical view of data, thereby isolating them from data reorganization.
- **Data isolation**: giving the program exactly that subset of the data it needs, thereby minimizing error propagation.
- **Authorization**: hiding sensitive information from a program, its authors, and users.

As the database evolves, records and sets are often “reorganized". Changing the underlying data should not cause all the programs to be recompiled or rewritten so long as the semantics of the data is not changed. Old programs should be able to see the data in the old way. Views are used to achieve this.

Typical reorganization operations include:

- Adding fields to records.
- Splitting records.
- Combining records.
- Adding or dropping access paths.

Simple view of base records may be obtained by:

- Renaming or permuting fields,
- Converting the representation of a field.

Simple variations of base sets may be obtained by:

- Selecting that subset of the records of a set which satisfy some predicate;
- Projecting out some fields or records in the set.
- Combining existing sets together into new virtual sets that can be viewed as a single larger set.

Consider the example of a set of records of the form:

```
+------+---------+------------------+----------------+
| NAME | ADDRESS | TELEPHONE_NUMBER | ACCOUNT_NUMBER |
+------+---------+------------------+----------------+
```

Some applications might be only interested in the name and telephone number, others might want name and address while others might want name and account number, and of course one application would like to see the whole record. A view can appropriately subset the base set if the set owner decides to partition the record into two new record sets:

```
PHONE_BOOK                                    ACCOUNTS
+------+---------+--------------+             +-------+----------------+
| NAME | ADDRESS | PHONE_NUMBER |             | NAME  | ACCOUNT_NUMBER |
+------+---------+--------------+             +-------+----------------+
```

Programs that used views will now access base sets (records) and programs that accessed the entire larger set will now access a view (logical set/record). This larger view is defined by:

```
DEFINE VIEW WHOLE_THING:
     SELECT NAME, ADDRESS, PHONE_NUMBER, ACCOUNT_NUMBER
     FROM PHONE_BOOK, ACCOUNTS
     WHERE PHONE_BOOK.NAME = ACCOUNTS.NAME;
```

3.5.1 Views and Update

Any view can support read operations; however, since only base sets are actually stored, only base sets can actually be updated. To make an update via a view, it must be possible to propagate the updates down to the underlying base set.

If the view is very simple (e. g., record subset) then this propagation is straightforward. If the view is a one-to-one mapping of records in some base set but some fields of the base are missing from the view, then update and delete present no problem but insert requires that the unspecified ("invisible") fields of the new records in the base set be filled in with the “undefined” value. This may or may not be allowed by the integrity constraints on the base set.

Beyond these very simple rules, propagation of updates from views to base sets becomes complicated, dangerous, and sometimes impossible.

To give an example of the problems, consider the WHOLE_THING view mentioned above. Deletion of a record may be implemented by a deletion from one or both of the constituent sets (PHONE_BOOK and ACCOUNTS). The correct deletion rule is dependent on the semantics of the data. Similar comments apply to insert and update.

My colleagues and I have resigned ourselves to the idea that there is no elegant solution to the view update problem. (Materialization (reading) is not a problem!) Existing systems use either very restrictive view mechanisms (subset only), or they provide incredibly ad hoc view update facilities. We propose that simple views (subsets) be done automatically and that a technique akin to that used for abstract data types be used for complex views: the view definer will specify the semantics of the operators NEXT, FETCH, INSERT, DELETE, and UPDATE,

3.6. STRUCTURE OF DATA MANAGER

Data manager is large enough to be subdivided into several components:

- **View component**: is responsible for interpreting the request, and calling the other components to do the actual work. The view component implements cursors and uses them to communicate as the internal and external representation of the view.
- **Record component**: stores logical records on “pages”, manages the contents of pages and the problems of variable length and overflow records.
- **Index component**: implements sequential and associative access to sets. If only associative access is required, hashing should be used. If both sequential and associative accesses are required then indices implemented as B-trees should be used (see Knuth Vol. 3 or IBM’s Virtual Sequential Access Method.)

- **Buffer manager**: maps the data "pages" on secondary storage to a primary storage buffer pool. If the operating system provided a really fancy page manager (virtual memory) then the buffer manager might not be needed. But, issues such as double buffering of sequential I/O, Write Ahead Log protocol (see recovery section), checkpoint, and locking seem to argue against using the page managers of existing systems. If you are looking for a hard problem, here is one: define an interface to page management that is useable by data management in lieu of buffer management.

3.7. A SAMPLE DATA BASE DESIGN

The introduction described a very simple database and a simple transaction that uses it. We discuss how that database could be structured and how the transaction would access it.

The database consists of the records

```
ACCOUNT(ACCOUNT_NUMBER, CUSTOMER_NUMBER, ACCOUNT_BALANCE, HISTORY)
CUSTOMER(CUSTOMER_NUMBER, CUSTOMER_NAME, ADDRESS,.....)
HISTORY(TIME, TELLER, CODE, ACCOUNT_NUMBER, CHANGE, PREV_HISTORY)
CASH_DRAWER(TELLER_NUMBER, BALANCE)
BRANCH_BALANCE(BRANCH, BALANCE)
TELLER(TELLER_NUMBER, TELLER_NAME,......)
```

This is a very cryptic description that says that a customer record has fields giving the customer number, customer name, address and other attributes.

The CASH_DRAWER, BRANCH_BALANCE and TELLER files (sets) are rather small (less than 100,000 bytes). The ACCOUNT and CUSTOMER files are large (about 1,000, 000,000 bytes). The history file is extremely large. If there are fifteen transactions against each account per month and if each history record is fifty bytes then the history file grows 7,500,000,000 bytes per month. Traffic on BRANCH_BALANCE and CASH_DRAWER is high and access is by BRANCH_NUMBER and TELLER_NUMBER respectively. Therefore these two sets are kept in high-speed storage and are accessed via a hash on these attributes. Traffic on the ACCOUNT file is high but random. Most accesses are via ACCOUNT_NUMBER but some are via CUSTOMER_NUMBER. Therefore, the file is hashed on ACCOUNT_NUMBER (partitioned set). A key-sequenced index, NAMES, is maintained on these records that gives a sequential and associative access path to the records ascending by customer name. CUSTOMER is treated similarly (having a hash on customer number and an index on customer name.) The TELLER file is organized as a sequential set. The HISTORY file is the most interesting. These records are written once and thereafter are only read. Almost every transaction generates such a record and for legal reasons the file must be maintained forever. This causes it to be kept as an entry sequenced set. New records are inserted at the end of the set. To allow all recent history records for a specific account to be quickly located, a parent child set is defined to link each ACCOUNT record (parent) to its HISTORY records (children). Each ACCOUNT record points to its most recent HISTORY record. Each HISTORY record points to the previous history record for that ACCOUNT.

Given this structure, we can discuss the execution of the DEBIT_CREDIT transaction outlined in the introduction. We will assume that the locking is done at the granularity of a page and that recovery is achieved by keeping a log (see section on transaction management.)

At initiation, the data manager allocates the cursors for the transaction on the ACCOUNTS, HISTORY, BRANCH, and CASH_DRAWER sets. In each instance it gets a lock on the set to insure that the set is available for update (this is an IX mode lock as explained in the locking section 5.7.6.2.) Locking at a finer granularity will be done during transaction execution (see locking section). The first call the data manager sees is a request to find the ACCOUNT record with a given account number. This is done by hashing the account number, thereby computing an anchor for the hash chain. Buffer manager is called to bring that page of the file into fast storage. Buffer manager looks in the buffer pool to see if the page is there. If the page is present, buffer manager returns it immediately. Otherwise, it finds a free buffer page slot, reads the page into that buffer and returns the filled buffer slot. Data manager then locks the page in share mode (so that no one else modifies it). This lock will be held to the end of the transaction. The record component searches the page for a record with that account number. If the record is found, its value is returned to the caller and the cursor is left pointing at the record.

The next request updates account balance of the record addressed by the cursor. This requires converting the share mode lock acquired by the previous call to a lock on the page in exclusive mode, so that no one else sees the new account balance until the transaction successfully completes. Also the record component must write a log record that allows it to undo or redo this update in case of a transaction or system abort (see section on recovery). Further, the transaction must note that the page depends on a certain log record so that buffer manager can observe the write ahead log protocol (see recovery section.)Lastly, the record component does the update to the balance of the record.

Next the transaction fabricates the history record and inserts it in the history file as a child of the fetched account. The record component calls buffer manager to get the last page of the history file (since it is an entry sequence set the record goes on the last page.) Because there is a lot of insert activity on the HISTORY file, the page is likely to be in the buffer pool. So buffer manager returns it, the record component locks it, and updates it. Next, the record component updates the parent-child set so that the new history record is a child of the parent account record. All of these updates are recorded in the system log in case of error.

The next call updates the teller cash drawer. This requires locking the appropriate CASH_DRAWER record in exclusive mode (it is located by hash). An undo-redo log record is written and the update is made.

A similar scenario is performed for the BRANCH_BALANCE file. When the transaction ends, data manager releases all its locks and puts the transaction's pages in the buffer manager's chain of pages eligible for write to disk.

If data manager or any other component detects an error at any point, it issues an ABORT_TRANSACTION command, which initiates transaction undo (see recovery section.)This causes data manager to undo all its updates to records on behalf of this user and then to release all its locks and buffer pages.

The recovery and locking aspects of data manager are elaborated in later sections. I suggest the reader design and evaluate the performance of the MONTHLY_STATATEMENT transaction described in the introduction as well as a transaction which given two dates and an account number will display the history of that account for that time interval.

3.8. COMPARISON TO FILE ACCESS METHODS

From the example above, it should be clear that data manager is a lot more fancy than the typical file access methods (indexed sequential files). File systems usually do not support partitioned or parent-child sets. Some support the notion of record, but none support the notions of field, network or view. They generally lock at the granularity of a file rather than at the granularity of a record. File systems generally do recovery by taking periodic image dumps of the entire file. This does not work well for a transaction environment or for very large files.

In general, data manager builds upon the operating system file system so that

- o The operating system is responsible for device support.
- o The operating system utilities for allocation, import, export and accounting are useable.
- o The data is available to programs outside of the data manager.

3.9. BIBLIOGRAPHY

Chamberlin et. al., “Views, Authorization, and Locking in a Relational Data Base System", 1975 NCC, Spartan Press. 1975. (Explains what views are and the problems associated with them.)

Computing Surveys, Vol. 8 No. 1, Barth 1976. (A good collection of papers giving current trends and issues related to the data management component of data base systems.)

Date, *Introduction to Database Systems*, Addison Wesley, 1975. (The seminal book on the data management part of data Management systems.)

Date, "An Architecture for High Level Language Database Extension;," Proceedings of 1976 SIGMOD Conference, ACM, 1976. (Unifies the relational, hierarchical and network models.)

Knuth, The Art of Computer Programming: Sorting and Searching, Vol. 3, Addison Wesley, 1975. (The seminal data structures book. Explains all about B-trees among other things.)

McGee, IBM Systems Journal, Vol. 16, No. 2, 1977, pp-84-160. (A very readable tutorial on IMS, what it does, how it works, and how it is used.)

Senko, "Data Structures and Data Accessing in Data Base Systems, Past, Present, Future," IBM Systems Journal, Vol. 16, No. 3, 1977, pp. 208-257. (A short tutorial on data models.)

4. DATA COMMUNICATION

The area of data communications is the least understood aspect of DB/DC systems. It must deal with evolving network managers, evolving intelligent terminals and in general seems to be in a continuing state of chaos. Do not feel too bad if you find this section bewildering.

Data communications is responsible for the flow of messages. Messages may come via telecommunications lines from terminals and from other systems, or messages may be generated by processes running within the system. Messages may be destined for external endpoints, for buffer areas called queues, or for executing processes. Data communications externally provides the functions of:

- Routing messages.
- Buffering messages,
- Message mapping so sender and receiver can each be unaware of the physical characteristics of the other.

Internally data communications provides:

- Message transformation that maps “external” messages to and from a format palatable to network manager.
- Device control of terminals.
- Message recovery in the face of transmission errors and system errors.

4.1. MESSAGES, SESSIONS, and RELATIONSHIP TO NETWORK MANAGER

Messages and endpoints are the fundamental objects of data communications. A message consists of a set of records. Records in turn consist of a set of fields. Messages therefore look very much like database sequential sets. Messages are defined by message descriptors. Typical unformatted definitions might be: A line from typewriter terminal is a one field, one record message. A screen image for a display is a two field (control and data), one record message. A multi-screen display image is a multi-field multi-record message.

Data communications depends heavily on the network manager provided by the base operating system. ARPANET, DECNET, and SNA (embodied in NCP and VTAM) are examples of such network managers. The network manager provides the notion of endpoint that is the smallest addressable network unit. A workstation, a queue, a process, and a card reader are each examples of endpoints.

Network manager transmits rather stylized transmission records (TRs).

These are simply byte strings. Network manager makes a best effort to deliver these byte strings to their destination. It is the responsibility of data communications to package messages (records and fields) into transmission records and then reconstructs the message from transmission records when they arrive at the other end.

The following figure summarizes this: application and terminal control programs see messages via sessions. DC in the host and terminal map:

these messages into transmission records that are carried by the network manager.

```
+---------------------+                 +---------------------+
| TERMINAL CONTROL    |                 |     TRANSACTION     |
|     PROGRAM         |                 |                     |
+---------------------+                 +---------------------+
      message                                  message
+---------------------+                 +---------------------+
|  DC IN TERMINAL     | <---session---> |     DC IN HOST      |
+---------------------+                 +---------------------+
  transmission record                     transmission record
+---------------------+                 +---------------------+
|    NETWORK MANAGER  |<-connection-->  | NETWORK MANAGER     |
+---------------------+                 +---------------------+
```

The three main layers of a session.

There are two ways to send messages: A one shot message can be sent to an endpoint in a canonical form with a very rigid protocol. Logon messages (session initiation) are often of this form. The second way to send messages is via an established session between the two endpoints. When the session is established, certain protocols are agreed to (e. g. messages will be recoverable (or not), session is half or full duplex, …) Thereafter, messages sent via the session these protocols. Sessions:

- o Establish the message formats desired by the sender and receiver.
- o Allow sender and receiver to validate one another's identity once rather than revalidating each message.
- o Allow a set of messages to be related together (see conversations).
- o Establish recovery, routing, and pacing protocols.

The network operating system provides connections between endpoints. A connection should be thought of as a piece of wire that can carry messages blocked (by data communications) into transmission records. Sessions map many to one onto connections. At any instant, a session uses a particular connection. But if the connection fails or if an endpoint fails, the session may be transparently mapped to a new connection. For example, if a terminal breaks, the operator may move the session to a new terminal. Similarly, if a connection breaks, an alternate connection may be established. Connections hide the problems of transmission management (an SNA term):

- o **Transmission control**, blocking and de-blocking transmission records (TRs), managing TR sequence numbers, and first-level retry logic.
- o **Path control**, or routing of TRs through the network.
- o **Link control**, sending TRs over teleprocessing lines.
- o **Pacing**, dividing the bandwidth and buffer pools of the network among connections.

The data communications component and the network manager cooperate in implementing the notion of session.

4.2. SESSION MANAGEMENT

The principal purpose of the session notion is:

- **Device independence**: the session makes transparent whether the endpoint is ASCII, EBCDIC, one-line, multi-line, program or terminal.
- **Abstraction**: manages the high level protocols for recovery, related messages and conversations.

Session creation specifies the protocols to be used on the session by each participant. One participant may be an ASCII typewriter and the other participant may be a sophisticated system. In this case the sophisticated system has lots of logic to handle the session protocol and errors on the session. On the other hand if the endpoint is an intelligent terminal and if the other endpoint is willing to accept the terminals protocol the session management is rather simple. (Note: The above is the way it is supposed to work. In practice sessions with intelligent terminals are very complex and the programs are much more subtle because intelligent terminals can make such complex mistakes. Typically, it is much easier to handle a master-slave session than to handle a symmetric session.)

Network manager simply delivers transmission records to endpoints. So, it is the responsibility of data communications to “know” about the device characteristics and to control the device. This means that DC must implement all the code to provide the terminal appearance. There is a version of this code for each device type (display, printer, typewriter,...), This causes the DC component to be very big in terms of thousands (K) Lines Of Code (KLOC).

If the network manager defines a generally useful endpoint model, then the DC manager can use this model for endpoints that fit the model. This is the justification for the TYPE1, TYPE2,... (endpoints) of SNA, and justifies the attempts to define a network logical terminal for ARPANET.

Sessions with dedicated terminals and peer nodes of the network are automatically (re)established when the system is (re)started. Of course, the operator of the terminal must re-establish his identity so that security will not be violated. Sessions for switched lines are created dynamically as the terminals connect to the system.

When DC creates the session, it specifies what protocols are to be used to translate message formats so that the session user is not aware of the characteristics of the device at the other end point.

4.3. QUEUES

As mentioned before a session may be:

FROM a program or terminal or queue

TO a program or terminal or queue

Queues allow buffered transmission between endpoints. Queues are associated (by DC) with users, endpoints and transactions. If a user is not logged on or if a process is doing something else, a queue can be used to hold one or more messages thereby freeing the session for further work or for termination. At a later time the program or endpoint may poll the queue and obtain the message.

Queues are actually passive so one needs to associate an algorithm with a queue. Typical algorithms are:

- o Allocate N servers for this queue,
- o Schedule a transaction when a message arrives in this queue.
- o Schedule a transaction when N messages appear in the queue.
- o Schedule a transaction at specified intervals.

Further, queues may be declared to be recoverable in which case DC is responsible for reconstructing the queue and it's messages if the system crashes or if the message consumer aborts.

4.4. MESSAGE RECOVERY

A session may be designated as recoverable in which case, all messages traveling on the session are sequence numbered and logged. If the transmission fails (positive acknowledge not received by sender), then the session endpoints resynchronize back to that message sequence number and the lost and subsequent messages are re-presented by the sender endpoint. If one of the endpoints fails, when it is restarted the session will be reestablished and the communication resumed. This requires that the endpoints be “recoverable” although one endpoint of a session may assume recovery responsibility for the other.

If a message ends up in a recoverable queue then:

- o If the dequeuer of the message (session or process) aborts, the message will be replaced in the queue.
- o If the system crashes, the queue will be reconstructed (using the log or some other recovery mechanism).

If the session or queue is not recoverable, then the message may be lost if the transmission, dequeuer or the system fails.

It is the responsibility of the data communications component to assure that a recoverable message is “successfully” processed or presented exactly once. It does this by requiring that receipt of recoverable messages acknowledged. A transaction "acknowledges” receipt of a message after it has processed it, at commit time (see recovery section.)

4.5. RESPONSE-MODE PROCESSING.

The default protocol for recoverable messages consists of the scenario:

```
TERMINAL                                 SYSTEM
           --------request----->
                                         log input message (forced)
           <-------acknowledge--
                                         process message
                                         commit (log reply forced)
           <-------reply-------
           --------acknowledge->
                                         log reply acknowledged
```

This implies four entries to the network manager (at each endpoint).

Each of these passes requires several thousand instructions (in typical implementations.) If one is willing to sacrifice the recoverability of the input message then the logging and acknowledgment of the input message can be eliminated and the reply sequence used as acknowledgment. This reduces line traffic and interrupt handling by a factor of two. This is the response mode message processing. The output (commit) message is logged. If something goes wrong before commit, it is as though the message was never received. If something goes wrong after commit then the log is used to re-present the message. However, the sender must be able to match responses to requests (he may send five requests and get three responses). The easiest way to do this is to insist that there is at most one message outstanding at a time (i.e. lock the keyboard).

Another scheme is to acknowledge a batch of messages with a single acknowledge. One does this by tagging the acknowledge with the sequence number of the latest message received. If messages are recoverable, then the sender must retain the message until it is acknowledged and so acknowledges should be sent fairly frequently.

4.5. CONVERSATIONS

A conversation is a sequence of messages. Messages are usually grouped for recovery purposes so that all are processed or ignored together. A simple conversation might consist of a clerk filling out a form. Each line the operator enters is checked for syntactic correctness and checked to see that the airline flight is available or that the required number of widgets is in stock. If the form passes the test it is redisplayed by the transaction with the unit price and total price for the line item filled in. At any time the operator can abort the conversation. However, if the system backs up the user (because of deadlock) or if the system crashes then when it restarts it would be nice to re-present the message of the conversation so that the operator’s typing is saved. This requires that the group of messages be identified to the data communications component as a conversation so that it can manage this recovery process. (The details of this protocol are an unsolved problem so far as I know.)

4.6. MESSAGE MAPPING

One of the features provided by DC is to insulate each endpoint from the characteristics of the other. There are two levels of mapping to do this: One level maps transmission records into messages. The next level maps the message into a structured message. The first level of mapping is defined by the session; the second level of mapping is defined by the recipient (transaction). The first level of mapping converts all messages into some canonical form (e. g. a byte string of EBCDIC characters.) This transformation may handle such matters as pagination on a screen (if the message will not fit on one screen image). The second level of mapping transforms the message from an uninterpreted string of bytes into a message-record-field structure. When one writes a transaction, one also writes a message mapping description that makes these transformations. For example, an airlines reservation transaction might have a mapping program that first displays a blank ticket. On input, the mapping program extracts the fields entered by the terminal operator and puts them in a set of (multi-field) records. The transaction reads these records (much in the style database records are read) and then puts out a set of records to be displayed. The mapping program fills in the blank ticket with these records and passes the resulting byte string to session management.

4.7. TOPICS NOT COVERED

A complete discussion of DC should include:

- o More detail on network manager (another lecturer will cover that).
- o Authorization to terminals (another lecturer will cover that).
- o More detail on message mapping.
- o The Logon-Signon process.

4.8. BIBLIOGRAPHY

Kimbelton, Schneider, "Computer Communication Networks: Approaches, Objectives, and Performance Considerations," *Computing Surveys*, Vol. 7, No. 3, Sept. 1975, (A survey paper on network managers.)

"Customer Information Control System/Virtual Storage (CICS/VS), System/Application Design Guide." IBM, form number SC33-3068, 1977 (An eminently readable manual on all aspects of data management systems. Explains various session management protocols and explains a rather nice message mapping facility.)

Eade, Homan, Jones, "CICS/VS and its Role in Systems Network Architecture," IBM Systems Journal, Vol. 16, No. 3, 1977 (Tells how CICS joined SNA and what SNA did for it.)

IBM Systems Journal, Vol. 15, No. 1, Jan. 1976. (All about SNA, IBM's network manager architecture.)

5. TRANSACTION MANAGEMENT

The transaction management system is responsible for scheduling system activity, managing physical resources, and managing system shutdown and restart. It includes components that perform scheduling, recovery, logging, and locking.

In general transaction management performs those operating system functions not available from the basic operating system. It does this either by extending the operating system objects (e. g. enhancing processes to have recovery and logging) or by providing entirely new facilities (e-g, independent recovery management.) As these functions become better understood, the duties of transaction management will gradually migrate into the operating system.

Transaction management implements the following objects:

Transaction descriptor: A transaction descriptor is a prototype for a transaction giving instructions on how to build an instance of the transaction. The descriptor describes how to schedule the transaction, what recovery and locking options to use, what data base views the transaction needs, what program the transaction runs, and how much space and time it requires.

Process: A process (domain) that is capable of running or is running a transaction. A process is bound to a program and to other resources. A process is a unit of scheduling and resource allocation. Over time a process may execute several transaction instances although at any instant a process is executing on behalf of at most one transaction instance. Conversely, a transaction instance may involve several processes. Multiple concurrent processes executing on behalf of a single transaction instance are called cohorts. Data management system processes are fancier than operating system processes since they understand locking, recovery and logging protocols but we will continue to use the old (familiar name for them).

Transaction instance: A process or collection of processes (cohorts) executing a transaction. A transaction instance is the unit of locking and recovery.

In what follows, we shall blur these distinctions and generically call each of these objects transactions unless a more precise term is needed.

The life of a transaction instance is fairly simple-A message or request arrives which causes a process to be built from the transaction descriptor. The process issues a BEGIN_TRANSACTION action that establishes a recovery unit. It then issues a series of actions against the system state. Finally it issues the COMMIT_TRANSACTION action that causes the outputs of the transaction to be made public (both updates and output messages.) Alternatively, if the transaction runs into trouble, it may issue the ABORT TRANSACTION action which cancels all actions performed by this transaction.

The system provides a set of objects and actions on these objects along with a set of primitives that allow groups of actions to be collected into atomic transactions. It guarantees no consistency on the objects beyond the atomicity of the actions. That is, an action will either successfully complete or it will not modify the system state at all.

Further, if two actions are performed on an object then the result will be equivalent to the serial execution of the two actions. (As explained below this is achieved by using locking within system actions.)

The notion of transaction is introduced to provide a similar abstraction above the system interface. Transactions are an all or nothing thing, either they happen completely or all trace of then (except in the log) is erased.

Before a transaction completes, it may be aborted and its updates to recoverable data may be undone. The abort can cone either from the transaction itself (suicide: bad input data, operator cancel,….) or from outside (murder: deadlock, timeout, system crash....) However, once a transaction commits (successfully completes), the effects of the transaction cannot be blindly undone. Rather, to undo a committed transaction, one must resort to compensation - running a new transaction that corrects the errors of its predecessor. Compensation is usually highly application dependent and is not provided by the system.

These definitions may be clarified by a few examples. The following is a picture of the three possible destinies of a transaction.

```
 BEGIN              BEGIN                        BEGIN
 action             action                       action
 action             action                       action
    .                 .            ABORT =>      action
    .                 .
    .               ABORT
 action
 COMMIT
A successful       A suicidal                 A murdered
Transaction        transaction                transaction
```

A simple transaction takes in a single message does something, and then produces a single message. Simple transactions typically make fifteen data base calls. Almost all transactions are simple at present (see Guide/Share Profile of IMS users). About half of all simple transactions are read-only (make no changes to the database.) For simple transactions, the notion of process, recovery unit and message coincide.

If a transaction sends and receives several synchronous messages it is called a conversational transaction. A conversational transaction has several messages per process and transaction instances. Conversational transactions are likely to last for a long time (minutes while the operator thinks and types) and hence pose special resource management problems.

The term batch transaction is used to describe a transaction that is “unusually big". In general such transactions are not on-line, rather they are usually started by a system event (timer driven) and run for a long time as a “background” job. Such a transaction usually performs thousands of data management calls before terminating. Often, the process will commit some of its work before the entire operation is complete. This is an instance of multiple (related) recovery units per process.

If a transaction does work at several nodes of a network then it will require a process structure (cohort) to represent its work at each participating node. Such a transaction is called distributed.

The following table summarizes the possibilities and shows the independence of the notions of process, message and transaction instance (commit). Cohorts communicate with one another via the session-message facilities provided by data communications.

	PROCESSES	MESSAGES	COMMITS
SIMPLE	1	1 in 1 out	1
CONVERSATIONAL	1	many in many out	1
BATCH	1	none(?)	many
DISTRIBUTED	many	1 in 1 out many among cohorts	1

We introduce an additional notion of save point in the notion of transaction. A save point is a firewall that allows a transaction to stop short of total backup. If a transaction gets into trouble (e. g. deadlock, resource limit) it may be sufficient to back up to such an intermediate save point rather than undoing all the work of the transaction. For example a conversational transaction which involves several user interactions might establish a save point at each user message thereby minimizing retyping by the user. Save points do not commit any of the transaction's updates. Each save point is numbered, the beginning of the transaction is save point 1 and successive save points are numbered 2, 3. . . . . The user is allowed to save some data at each save point and to retrieve this data if he returns to that point. Backing up to save point 1 resets the transaction instance to the recovery component provides the actions:

o BEGIN_TRANSACTION: designates the beginning of a transaction.

o SAVE_TRANSACTION: designates a firewall within the transaction.

If an incomplete transaction is backed-up, undo may stop at such a point rather than undoing the entire transaction,

o BACKUP_TRANSACTION: undoes the effects of a transaction to an earlier save point.

o COMMIT_TRANSACTION: signals successful completion of transaction and causes outputs to be committed.

o ABORT_TRANSACTION: causes undo of a transaction to its initial state.

Using these primitives, application programs can construct groups of actions that are atomic. It is interesting that this one level of recovery is adequate to support multiple levels of transactions by using the notion of save point.

The recovery component supports two actions that deal with system recovery rather than transaction recovery:

o CHECKPOINT: Coordinates the recording of the system state in the log.

o RESTART: Coordinates system restart, reading the checkpoint log record and using the log to redo committed transactions and to undo transactions that were uncommitted at the time of the shutdown or crash.

5.1. TRANSACTION SCHEDULING

The scheduling problem can be broken into many components: listening for new work, allocating resources for new work, scheduling (maintaining the dispatcher list), and dispatching.

The listener is event driven. It receives messages from data communications and from dispatched processes.

A distinguished field of the message specifies a transaction name. Often, this field has been filled in by data communications that resolved the transaction name to a reference to a transaction descriptor. Sometimes this field is symbolic in which case the listener uses the name in a directory call to get a reference to the transaction descriptor, (The directory may be determined by the message source.) If the name is had or if the sender is not authorized to invoke the transaction then the message is discarded and a negative acknowledge is sent to the source of the message.

If the sender is authorized to invoke the named transaction, then the allocator examines the transaction descriptor and the current system state and decides whether to put this message in a work-to-do list or to allocate the transaction right away, Criteria for this are:

- The system may be overloaded (" full".)
- There may be a limit on the number of transactions of this type, which can run concurrently.
- There may be a threshold, N, such that N messages of this type must arrive, at which point a server is allocated and the messages are batched to this server.
- The transaction may have an affinity to resources, which are unavailable.
- The transaction may run at a special time (overnight, off-shift,...)

If the transaction can run immediately, then the allocator either allocates a new process to process the message or gives the message to a primed transaction that is waiting for input.

If a new process is to be created, a process (domain) is allocated and all objects mentioned in the transaction descriptor are allocated as part of the domain, Program management sets up the address space to hold the programs, data management will allocate the cursors of the transaction for the process, data communication allocates the necessary queues, the recovery component allocates a log cursor and writes a begin transaction log record, and so on. The process is then set up with a pointer to the input message.

This allocated process is given to the scheduler that eventually places it on the dispatcher queue. The dispatcher eventually runs the process.

once the transaction scheduler dispatches the process, the operating system scheduler is responsible for scheduling the process against the physical resources of the system.

When the transaction completes, it returns to the scheduler. The scheduler may or may not collapse the process structure depending on whether the transaction is batched or primed. If the transaction has released resources needed by waiting unscheduled transactions, the scheduler will now dispatch these transactions.

Primed transactions are an optimization that dramatically reduce allocation and deallocation overhead. Process allocation can be an expensive operation and so transactions that are executed frequently are often primed. A primed transaction has a large part of the domain already built. In particular programs are loaded, cursors are allocated and the program prolog has been executed. The transaction (process) is waiting for input. The scheduler need only pass the message to the transaction (process), Often the system administrator or operator will prime several instances of a transaction. A banking system doing three withdrawals and five deposits per second might have two withdrawal transactions and four deposit transactions primed.

Yet another variant has the process ask for a message after it completes. If a new message has arrived for that transaction type, then the process processes it. If there is no work for the transaction, then the process disappears. This is called batching messages as opposed to priming. It is appropriate if message traffic is “bursty” (not uniformly distributed in time). It avoids keeping a process allocated when there is no work for it to do.

5.2. DISTRIBUTED TRANSACTION MANAGEMENT

A distributed system is assumed to consist of a collection of autonomous nodes that are tied together with a distributed data communication system in the style of high level ARPANET, DECNET, or SNA protocols. Resources are assumed to be partitioned in the sense that a resource is owned by only one node. The system should be:

- Inhomogeneous (nodes are small, medium, large, ours, theirs,...)
- Unaffected by the loss of messages.
- Unaffected by the loss of nodes (i.e. requests to that node wait for the node to return, ether nodes continue working.)

Each node may implement whatever data management and transaction management system it wants to. We only require that it obey the network protocols, Some node might be a minicomputer running a fairly simple data management system and using an old-master new-master recovery protocol. Another node might be running a very sophisticated data management system with many concurrent transactions and fancy recovery.

If one transaction may access resources in many nodes of a network then a part of the transaction must "run” in each node. We already have an entity that represents transaction instances: processes.

Each node will want to

- o Authorize local actions of the process (transaction).
- o Build an execution environment for the process (transaction).
- o Track local resources held by the process (transaction).
- o Establish a recovery mechanism to undo the local updates of that process (see recovery section).
- o Observe the two-phase commit protocol (in cooperation with its cohorts (see section on recovery)).

Therefore, the structure needed for a process in a distributed system is almost identical to the structure needed by a transaction in a centralized system.

This latter observation is key. That is why I advocate viewing each node as a transaction processor. (This is a minority view.) To install a distributed transaction, one must install prototypes for its cohorts in the various nodes. This allows each node to control access by distributed transactions in the same way it controls access by terminals. If a node wants to give away the keys to its kingdom it can install a universal cohort (transaction) which has access to all data and which performs all requests.

If a transaction wants to initiate a process (cohort) in a new node, some process of the transaction must request that the node construct a cohort and that the cohort go into session with the requesting process (see data communications section for a discussion of sessions). The picture below shows this.

```
NODE1
+------------+
|  ********  |
|  * T1P2 *  |
|  ********  |
|     #      |
+-----#------+
      #
 +----#----+
 | SESSION |
 +----#----+
      #
NODE2 #
+-----#------+
|     #      |
|  ********  |
|  * T1P2 *  |
|  ********  |
+------------+
```

Two cohorts of a distributed transaction in session.

A process carries both the transaction name T1 and the process name (in NODE1 the cohort of T1 is process P2 and in NODE2 the cohort of T1 is process P6.)

The two processes can now converse and carry out the work of the transaction. If one process aborts, they should both abort, and if one process commits they should both commit. Thus they need to:

- Obey the lock protocol of holding locks to end of transaction (see section on locking).
- Observe the two-phase commit protocol (see recovery section).

These comments obviously generalize to transactions of more than two charts.

5.3. THE DATA MANAGEMENT SYSTEM AS A SUBSYSTEM

It has been the recent experience of general-purpose operating systems that the operating system is extended or enhanced by some "application program" like a data management system, or a network management system. Each of these systems often has very clear ideas about resource management and scheduling. It is almost impossible to write such systems unless the basic operating system:

- allows the subsystem to appear to users as an extension of the basic operating system.
- allows the subsystem to participate in major system events such as system shutdown/restart, process termination,....

To cope with these problems, operating systems have either made system calls indistinguishable from other calls (e. g. MULTICS) or they have reserved a set of operating systems calls for subsystems (e. g. user SVCs in OS/360.) These two approaches address only the first of the two problems above.

The notion of subsystem is introduced to capture the second notion. For example, in IBM's operating system VS release 2.2, notifies each known subsystem at important system events (e. g, startup, memory failure, checkpoint,...) Typically a user might install a Job Entry Subsystem, a Network Subsystem, a Text Processing Subsystem, and perhaps several different Data Management Subsystems on the same operating system, The basic operating system serves as a coordinator among these sub-systems.

- It passes calls from users to these subsystems.
- It broadcasts events to all subsystems.

The data manager acts as a subsystem of the host extending its basic facilities.

The data management component is in turn comprised following is a partial list of the components in the bottom half of the data base component of System R:

- Catalog manager: maintains directories of system.
- Call analyzer: regulates system entry-exit.

- Record manager: extracts records from pages.
- Index component: maintains indices on the database.
- Sort component: maintains sorted versions of sets.
- Loader: performs bulk insertion of records into a file.
- Buffer manager: maps database pages to and from secondary storage.
- Performance monitor: Keeps statistics about system performance and state.
- Lock component: maintains the locks (synchronization primitives).
- Recovery manager: implements the notion of transaction COMMIT, ABORT, and handles system restart.
- Log manager: maintains the system log.

Notice that primitive forms of these functions are present in most general-purpose operating systems. In the future one may expect to see the operating system subsume most of these data management functions.

5.4. EXCEPTION HANDLING

The protocol for handling synchronous errors (errors which are generated by the process) is another issue defined by transaction management (extending the basic operating systems facilities). In general the data management system wants to abort the transaction if the application program fails. This is generally handled by organizing the exceptions into a hierarchy. If a lower level of the hierarchy fails to handle the error, it is passed to a higher node of the hierarchy. The data manager usually has a few handlers very near the top of the hierarchy (the operating system gets the root of the hierarchy.)

- Either the process or the data management system (or both) may ----establish an exception handler to field errors.
- When, an exception is detected then the exception is signaled.
- Exception handlers are invoked in some fixed order (usually order of establishment) until one successfully corrects the error. This operation is called percolation.

PL/I “ON units” or the IBM Operating System set-task-abnormal-exit (STAE) are instances of this mechanism. Examples of exception conditions are: arithmetic exception conditions (i.e., overflow), invalid program reference (i.e., to protected storage) wild branches, infinite loops, deadlock, .. and attempting to read beyond end of file.

There may be several exception handlers active for a process at a particular instant. The program's handler is usually given the first try at recovery if the program has established a handler. The handler will, in general, diagnose the failure as one that was expected (overflow), one that was unexpected but can be handled (invalid program reference), or one that is unexpected and cannot be dealt with by the handler (infinite loop). If the failure can be corrected, the handler makes the correction and continues processing the program (perhaps at a different point of execution.) If the failure cannot be corrected by this handler, then the exception will percolate to the next exception handler for that process.

The system generally aborts any process, which percolates to the system recovery routine or does not participate in recovery. This process involves terminating all processing being done on behalf of the process, restoring all non-consumable resources in use by the process to operating system control (i.e., storage), and removing to the greatest extent possible the effects of the transaction.

5.5. OTHER COMPONENTS WITHIN TRANSACTION MANAGEMENT

We mention in passing that the transaction management component must also support the following notions:

- Timer services: Performing operations at specified times. This involves running transactions at specified times or intervals and providing a timed wait if it is not available from the base operating system.
- Directory management: Management of the directories used by transaction management and other components of the system. This is a high-performance low-function in-core data management system. Given a name and a type (queue, transaction, endpoint,...)it returns a reference to the object of that name and type. (This is where the cache of dictionary descriptors is kept.)
- Authorization Control: Regulates the building and use of transactions.

These topics will be discussed by other lecturers.

5.6. BIBLIOGRAPHY.

Stonebraker, Neuhold, “A Distributed Data Base Version of INGRESS", Proceedings of Second Berkeley Workshop on Networks and Distributed Data, Lawrence Livermore Laboratory, (1977)-(Gives another approach to distributed transaction management.)

*'Information Management System/Virtual Storage (IBS/VS) System Manual Vol. 1: Logic. II, IBM, form number LY20-8004-2. (Tells all about IMS. The discussion of scheduling presented here is in the tradition of JMS/VS pp 3.36-3.41.)

"OS/W2 System Logic Library”, IBM, form number SY28-0763, (Documents the subsystem interface of OS/VS pp-3.159-168)

"OS/VS MVS Supervisor Services and Macro Instructions.", IBM, form number GC28-0756, (Explains percolation on pages 53-62.)

5.7. LOCK MANAGEMENT.

This section derives from papers co-authored with Irv Traiger and Franco Putzolu.

The system consists of objects that are related in certain ways. These relationships are best thought of as assertions about the objects. Examples of such assertions are:

“Names is an index for Telephone-numbers.”

“Count_of_x is the number of employees in department x.”

The system state is said to be consistent if it satisfies all its assertions. In some cases, the database must become temporarily inconsistent in order to transform it to a new consistent state. For example, adding a new employee involves several atomic actions and updating several fields. The database may be inconsistent until all these updates have been completed.

To cope with these temporary inconsistencies, sequences of atomic actions are grouped to form transactions. Transactions are the units of consistency. They are larger atomic actions on the system state that transform it from one consistent state to a new consistent state. Transactions preserve consistency. If some action of a transaction fails then the entire transaction is 'undone,' thereby returning the database to a consistent state. Thus transactions are also the units of recovery. Hardware failure, system error, deadlock, protection violations and program error are each a source of such failure.

5.7.1. PROS AND CONS OF CONCURRENCY

If transactions are run one at a time then each transaction will see the consistent state left behind by its predecessor. But if several transactions are scheduled concurrently then the inputs of some transaction may be inconsistent even though each transaction in isolation is consistent.

Concurrency is introduced to improve system response and utilization.

- It should not cause programs to malfunction.
- Concurrency control should not consume more resources than it “saves”.

If the database is read-only then no concurrency control is needed. However, if transactions update shared data then their concurrent execution needs to be regulated so that they do not update the same item at the same time.

If all transactions are simple and all data are in primary storage then there is no need for concurrency. However, if any transaction runs for a long time or does I/O then concurrency may be needed to improve responsiveness and utilization of the system. If concurrency is allowed, then long-running transactions will (usually) not delay short ones.

Concurrency must be regulated by some facility that regulates access to shared resources. Data management systems typically use locks for this purpose.

The simplest lock protocol associates a lock with each object. Whenever using the object, the transaction acquires the lock and holds it until the transaction is complete. The lock is a serialization mechanism that insures that only one transaction accesses the object at a time. It has the effect of: notifying others that the object is busy; and of protecting the lock requestor from modifications of others.

This protocol varies from the serially reusable resource protocol common to most operating systems (and recently renamed monitors) in that the lock protocol holds locks to transaction commit. It will be argued below that this is a critical difference.

Responsibility for requesting and releasing locks can either be assumed by the user or be delegated to the system. User controlled locking results in potentially fewer locks due to the user's knowledge of the semantics of the data. On the other hand, user controlled locking requires difficult and potentially unreliable application programming. Hence the approach taken by most data base systems is to use automatic lock protocols which insure protection from inconsistency, while still allowing the user to specify alternative lock protocols as an optimization.

5.7.2 CONCURRENCY PROBLEMS

Locking is intended to eliminate three forms of inconsistency due to concurrency.

- **Lost Updates**: If transaction T1 updates a record previously updated by transaction T2 then undoing T2 will also undo the update of T1 (i.e. if transaction T1 updates record R from 100 to 101 and then transaction T2 updates A from 101 to 151 then backing out T1 will set A back to the original value of 100 losing the update of T2.) This is called a Write ->Write dependency.
- **Dirty Read**: If transaction T1 updates a record that is read by T2, then if T1 aborts, T2 will have read a record that never existed. (i.e. T1 updates R to 100,000,000, T2 reads this value, T1 then aborts and the record returns to the value 100.) This is called a Write ->Read dependency.
- **Un-repeatable Read**: If transaction T1 reads a record that is then altered and committed by T2, and if T1 re-reads the record, then T1 will see two different committed values for the sane record. Such a dependency is called a Read ->Write dependency.

If there were no concurrency then none of these anomalous cases will arise.

Note that the order in which reads occur does not affect concurrency.

In particular reads commute. That is why we do not care about Read -> Read dependencies,

5.7.3. MODEL OF CONSISTENCY AND LOCK PROTOCOLS

A fairly formal model is required in order to make precise statements about the issues of locking and recovery. Because the problems are so complex one must either accept many simplifying assumptions or accept a less formal approach. A compromise is adopted here. First we will introduce a fairly formal model of transactions, locks and recovery that will allow us to discuss the issues of lock management and recovery management. After this presentation, the implementation issues associated with locking and recovery will be discussed.

5.7.3.1. SEVERAL DEFINITIONS OF CONSISTENCY

Several equivalent definitions of consistency are presented. The first definition is an operational and intuitive one; it is useful in describing the system behavior to users. The second definition is a procedural one in terms of lock protocols; it is useful in explaining the system implementation. The third definition is in terms of a trace of the system actions; it is useful in formally stating and proving consistency properties.

5.7.3.1.1. INFORMAL DEFINITION OF CONSISTENCY

An output (write) of a transaction is committed when the transaction abdicates the right to “undo” the write thereby making the new value available to all other transactions (i.e. commits). Outputs are said to be uncommitted or dirty if they are not yet committed by the writer. Concurrent execution raises the problem that reading or writing other transactions’ dirty data may yield inconsistent data.

Using this notion of dirty data, consistency may be defined as:

Definition 1: Transaction T sees a consistent state if:

(a) T does not overwrite dirty data of other transactions.

(b) T does not commit any writes until it completes all its writes (i.e. until the end of transaction (EOT)).

(c) T does not read dirty data from other transactions.

(d) Other transactions do not dirty any data read by T before T completes.

Clauses (a) and (b) insure that there are no lost updates.

Clause (c) isolates a transaction from the uncommitted data of other transactions. Without this clause, a transaction might read uncommitted values, which are subsequently updated or are undone. If clause (c)is observed, no uncommitted values are read.

Clause (a) insures repeatable reads. For example, without clause (c) a transaction may read two different (committed) values if it reads the same entity twice. This is because a transaction that updates the entity could begin, update, and commit in the interval between the two reads. More elaborate kinds of anomalies due to concurrency are possible if one updates an entity after reading it or if more than one entity is involved (see example below).

The rules specified have the properties that:

1. If all transactions observe the consistency protocols, then any execution of the system is equivalent to some “serial” execution of the transactions (i.e. it is as though there was no concurrency.)
2. If all transactions observe the consistency protocols, then each transaction sees a consistent state.
3. If all transactions observe the consistency protocols, then system backup (undoing all in-progress transactions) loses no updates of completed transactions.
4. If all transactions observe the consistency protocols, then transaction backup (undoing any in-progress transaction) produces a consistent state.

Assertions 1 and 2 are proved in the paper "On the Notions of Consistency and Predicate Locks" CACM Vol. 9, No. 71, Nov. 1976. Proving the second two assertions is a good research problem. It requires extending the model used for the first two assertions and reviewed here to include recovery notions.

5.7.3.1.2. SCHEDULES: FORMALIZE DIRTY AND COMMITTED DATA

The definition of what it means for a transaction to see a consistent state was given in terms of dirty data. In order to make the notion of dirty data explicit, it is necessary to consider the execution of a transaction in the context of a set of concurrently executing transactions. To do this we introduce the notion of a schedule for a set of transactions. A schedule can be thought of as a history or audit trail of the actions performed by the set of transactions. Given a schedule the notion of a particular entity being dirtied by a particular transaction is made explicit and hence the notion of seeing a consistent state is formalized. These notions may then be used to connect the various definitions of consistency and show their equivalence.

The system directly supports objects and actions. Actions are categorized as begin actions, end actions, abort actions, share lock actions, exclusive lock actions, unlock actions, read actions, write actions. Commit actions and abort actions are presumed to unlock any locks held by the transaction but not explicitly unlocked by the transaction. For the purposes of the following definitions, share lock actions and their corresponding unlock actions are additionally considered to be read actions and exclusive lock actions and their corresponding unlock actions are additionally considered to be write actions.

For the purposes of this mad, a transaction is any sequence of actions beginning until a begin action and ending with a commit or abort action and not containing other begin, commit or abort actions. Here are two trivial transactions.

```
T1 BEGIN                        T2 BEGIN
SHARE LOCK     A                SHARE LOCK B
EXCLUSIVE LOCK B                READ       B
READ           A                SHARE LOCX A
WRITE B                         READ       A
COMMIT                          ABORT
```

Any (sequence preserving) merging of the actions of a set of transactions into a single sequence is called a schedule for the set of transactions.

A schedule is a history of the order in which actions were successfully executed (it does not record actions which were undone due to backup (This aspect of the model needs to be generalized to prove assertions 3 and 4 above)). The simplest schedules run all actions of one transaction and then all actions of another transaction,... Such one-transaction-at-a-time schedules are called serial because they have no concurrency among transactions. Clearly, a serial schedule has no concurrency-induced inconsistency and no transaction sees dirty data. Locking constrains the set of allowed schedules. In particular, a schedule is legal only if it does not schedule a lock action on an entity for one transaction when that entity is already locked by some other transaction in a conflicting mode.

The following table shows the compatibility among the simple lock modes.

		LOCK MODE	
COMPATIBILITY		SHARE	EXCLUSIVE
REQUEST	SHARE	COMPATIBLE	CONFLICT
MODE	EXCLUSIVE	CONFLICT	CONFLICT

The following are three example schedules of two transactions. The first schedule is legal, the second is serial and legal and the third schedule is not legal since T1 and T2 have conflicting locks on the object A.

T1 BEGIN	T1 BEGIN	T2 BEGIN
T2 BEGIN	T1 SHARE LOCK A	T1 BEGIN
T2 SHARE LOCK B	T1 EXCLUSIVE LOCK B	T1 EXCLUSIVE LOCK A
T2 READ B	T1 READ A	T2 SHARE LOCK B
T1 SHARE LOCK A	T1 WRITE B	T2 READ B
T2 SHARE LOCK A	T1 COMMIT	T2 SHARE LOCK A
T2 READ A	T2 BEGIN	T2 READ A
T2 ABORT	T2 SHARE LOCK B	T2 ABORT
T1 EXCLUSIVE LOCK B	T2 READ B	T1 SHARE LOCK B
T1 READ A	T2 SHARE LOCK A	T1 READ A
T1 WRITE B	T2 READ A	T1 WRITE B
T1 COMMIT	T2 ABORT	T1 COMMIT
Legal & not serial	legal & serial	not legal& not serial

The three varieties of schedules (serial and not legal is impossible).

An initial state and a schedule completely define the system's behavior. At each step of the schedule one can deduce which entity values have been committed and which are dirty: if locking is used, updated data is dirty until it is unlocked.

One transaction instance is said to depend on another if the first takes some of its inputs from the second. The notion of dependency can be useful in comparing two schedules of the same set of transactions.

Each schedule, S, defines a ternary dependency relation on the set: TRANSACTIONS X OBJECTS X TRANSACTIONS as follows. Suppose that transaction T performs action a on entity e at some step in the schedule, and that transaction T' performs action a' on entity e at a later step in the schedule. Further suppose that T and T' are distinct.

Then:

(T, e, T') is in DEP(S)

if a is a write action and a' is a write action

or a is a write action and a' is a read action

or a is a read action and a' is a write action

The dependency set of a schedule completely defines the inputs and outputs each transaction "sees". If two distinct schedules have the same dependency set then they provide each transaction with the same inputs and outputs. Hence we say two schedules are equivalent if they have the same dependency sets. If a schedule is equivalent to a serial schedule, then that schedule must be consistent since in a serial schedule there are no inconsistencies due to concurrency. On the other hand, if a schedule is not equivalent to a serial schedule then it is probable (possible) that some transaction sees an inconsistent state. Hence,

Definition 2: A schedule is *consistent* if it is equivalent to some serial schedule.

The following argument may clarify the inconsistency of schedules not equivalent to serial schedules. Define the relation <<<on the set of transactions by:

T<<<T` 'if for some entity e (T, e, T') is in DEP(S).

Let <<<* be the transitive closure of <<<, then define:

BEFORE(T) = {T' I T'<<<*T }

AFTER(T) = { ~T' | T <<<*T' }.

The obvious interpretation of this is that BEFORE(T) is the set of transactions that contribute inputs to T and each AFTER(T) set is the set of transactions that take their inputs from T

If some transaction is both before T and after T in some schedule then no serial schedule could give such results. In this case, concurrency has introduced inconsistency. On the other hand, if all relevant transactions are either before or after T (but not both)then T will see a consistent state. If all transactions dichotomize others in this way then the relation <<<*will be a partial order and the whole schedule will provide consistency.

The above definitions can be related as follows:

<u>Assertion:</u>

A schedule is consistent
if and only if (by definition)
the schedule is equivalent to a serial schedule
if and only if
the relation <<<*is a partial order.

<u>5.7.3.1.3. Lock Protocol Definition of Consistency</u>

Whether an instantiation of a transaction sees a consistent state depends on the actions of other concurrent transactions. All transactions agree to certain lock protocols so that they can all be guaranteed consistency.

Since the lock system allows only legal schedules we want a lock protocol such that: every legal schedule is a consistent schedule.

Consistency can be procedurally defined by the lock protocol, which produces it: A transaction locks its inputs to guarantee their consistency and locks its outputs to mark them as dirty (uncommitted).

For this section, locks are dichotomized as <u>share mode</u> locks which allow multiple readers of the same entity and <u>exclusive mode</u> locks which reserve exclusive access to an entity.

The lock protocols refined to these modes is:

Definition 3: Transaction T observes the consistency lock protocol if:

(al T sets an exclusive lock on any data it dirties
(b) T sets a share lock on any data it reads.
(c) T holds all locks to end of transaction.

These lock protocol definitions can be stated more precisely and tersely with the introduction of the following notation. A transaction is well-formed if it always locks an entity in exclusive (shared or exclusive) mode before writing (reading) it.

A transaction is two-phase if it does not (share or exclusive) lock an entity after unlocking some entity, A two-phase transaction has a growing phase during which it acquires locks and a shrinking phase during which it releases locks.

The lock consistency protocol can be redefined as:

Definition 3’: Transaction T observes the consistency lock protocol if it is well formed and two phase.

Definition 3 was too restrictive in the sense that consistency does not require that a transaction hold all locks to the EOT (i.e. the ROT need not be the start of the shrinking phase). Rather, the constraint that the transaction be two-phase is adequate to insure consistency. On the other hand, once a transaction unlocks an updated entity, it has committed that entity and so cannot be undone without cascading backup to any transactions that may have subsequently read the entity. For that reason, the shrinking phase is usually deferred to the end of the transaction; thus, the transaction is always recoverable and all updates are committed together.

5.7.3.2. Relationship Among Definitions

These definitions may be related as follows: if T sees a consistent state in S.

Assertion:

(a) If each transaction observes the consistency lock protocol (Definition 3'), then any legal schedule is consistent (Definition 2) (i.e. each transaction sees a consistent state in the sense of Definition 1).

(b) If transaction T violates the consistency lock protocol then it is possible to define another transaction T’ that does observe the consistency lock protocol such that T and T' have a legal schedule S but T does not see a consistent state in S.

This says that if a transaction observes the consistency lock protocol definition of consistency (Definition 3') then it is assured of the definition of consistency based on committed and dirty data (Definition 1 or 3). Unless a transaction actually Sets the locks prescribed by consistency one can construct transaction mixes and schedules which will cause the transaction to see an inconsistent state. However, in particular cases such transaction mixes may never occur due to the structure or use of the system. In these cases an apparently inadequate locking may actually provide consistency. For example, a data base reorganization usually need no locking since it is run as an off-line utility which is never run concurrently with other transactions.

5.7.4. LOCKING, TRANSACTION BACKUP, AND SYSTEM RECOVERY

To repeat, there is no nice formal model of recovery (Lampson and Sturgis have a model in their forth-coming CACM paper on two-phase commit processing, but the model in the version I saw was rather vague.) Here, we will limp along with a (even more) vague model.

A transaction T is said to be recoverable if it can be undone before EOT without undoing other transactions' updates. A transaction T is said to be repeatable if it will reproduce the original output if rerun following recovery, assuming that no locks were released in the backup process. Recoverability requires update locks be held to commit point. Repeatability requires that all transactions observe the consistency lock protocol.

The normal (i.e. trouble free) operation of a data base system can be described in terms of an initial consistent state S0 and a schedule of transactions mapping the data base into a final consistent state S3 (see Figure). S1 is a checkpoint state, since transactions are in progress, S1 may be inconsistent. A system crash leaves the database in state S2. Since transactions T3 and T5 were in progress at the time of the crash. S2 is potentially inconsistent. System recovery amounts to bringing the database to a new consistent state in one of the following ways:

(a) Starting from state S2, undo all actions of transactions in-progress at the time of the crash.

(b) Starting from state S1 first undo all actions of transactions in progress at the time of the crash (i.e. actions of T3 before S1) and then redo all actions of transactions that completed after S1 and before the crash (i.e. actions of T2 and T4 after S1.)

(c) Starting at SO redo all transactions that completed before the crash.

Observe that (a) and (c)are degenerate cases of (b).

```
|                           |          <                  |
|   T1|------------|        |          >                  |
|         T2|---------------|---|      <                  |
|            T2|------------|---------->---|              |
|         T3|---------------|---|      <                  |
|                           |  T4|---| >                  |
|                           |   T5|----<----|             |
S0              (checkpoint)S1         S2 (crash)         S3 (recovered)
```

Figure. System states, SO is initial state, S1 is checkpoint state, S2 is a crash and S3 is the state that results in the absence of a crash.

If some transaction does not hold update locks to commit point then:

- o Undoing the transaction may deadlock (because undo must reacquire the locks in order to perform undo.)
- o Undoing a transaction may loose updates (because an update may have been applied to the output of the undone transaction, but backup will restore the entity to its original value.)

Consequently, backup may cascade: backing up one transaction may require backing up another. (Randall calls this the domino effect.) If for example, T3 writes a record, r, and then T4 further updates r then undoing T3 will cause the update of T4 to r to be lost. This situation can only arise if some transaction does not hold its write locks to commit point. For these reasons all known data management systems (which support concurrent updaters) require that all transactions hold their update locks to commit point.

On the other hand,

- If all the transactions hold all update locks to commit point, then system recovery loses no updates of complete transactions. However there may be no schedule that would give the same result because some transactions may have read outputs of undone transactions (dirty reads).

- If all the transactions observe the consistency lock protocol, then the recovered state is consistent and derives from the schedule obtained from the original system schedule by deleting incomplete transactions, Note that consistency prevents read dependencies on transactions which might be undone by system recovery. The schedule obtained by considering only the actions of completed transactions produces the recovered state.

Transaction crash gives rise to transaction backup that has properties analogous to system recovery.

5.7.5. LOWER DEGREES OF CONSISTENCY

Most systems do not provide consistency as outlined here. Typically they do not hold read locks to EOT so that R->W->R dependencies are not precluded. Very primitive systems sometimes set no read locks at all, rather they only set update locks so as to avoid lost update and deadlock during backout. We have characterized these lock protocols as degree 2 and degree 1 consistency respectively and have studied them extensively (see "Granularity of locks and degrees of consistency in a shared data base", Gray, Lorie, Putzolu, and Traiger. in *Modeling & Data Base Systems*, North Holland Publishing (1976).) I believe that the lover degrees of consistency are a bad idea, but several of my colleagues disagree. The motivation of the lower degrees is performance. If less is locked, then less computation and storage is consumed. Furthermore, if less is locked, concurrency is increased since fewer conflicts appear. (Note that minimizing the number of explicit locks set motivates the granularity lock scheme of the next section.)

5.7.5. LOCK GRANULARITY

An important issue that arises in the design of a system is the choice of lockable units (i.e. the data aggregates which are atomically locked to insure consistency.) Examples of lockable units are areas, files, individual records, field values, and intervals of field values.

The choice of lockable units presents a tradeoff between concurrency and overhead, which is related to the size or granularity of the units themselves. On the one hand, concurrency is increased if a fine lockable unit (for example a record or field) is chosen. Such unit is appropriate for a "simple" transaction that accesses few records. On the other hand, a fine unit of locking would be costly for a “complex” transaction that accesses many records. Such a transaction would have to set and reset many locks, incurring the computational overhead of many invocations of the lock manager, and the storage overhead of representing many locks. A coarse lockable unit (for example a file) is probably convenient for a transaction that accesses many records. However, such a coarse unit discriminates against transactions that only want to lock one member of the file. From this discussion it follows that it would be desirable to have lockable units of different granularities coexisting in the same system.

The following presents a lock protocol satisfying these requirements and discusses the related implementation issues of scheduling, granting and converting lock requests.

5.7.6.1. HIERARCHICAL LOCKS

We will first assume that the set of resources to be locked is organized in a hierarchy. Note that this hierarchy is used in the context of a collection of resources and has nothing to do with the data model used in a data base system. The hierarchy of the following figure may be suggestive. We adopt the notation that each level of the hierarchy is given a node type that is a generic name for all the node instances of that type. For example, the database has nodes of type area as its immediate descendants, each area in turn has nodes of type file as its immediate descendants and each file has nodes of type record as its immediate descendants in the hierarchy. Since it is a hierarchy, each node has a unique parent.

```
DATA BASE
    |
  AREAS
    |
  FILES
    |
 RECORDS
```

Figure 1: A sample lock hierarchy.

Each node of the hierarchy can be locked. If one requests exclusive access (X) to a particular node, then when the request is granted, the requestor has exclusive access to that node and implicitly to each of its descendants. If one requests shared access (S) to a particular node, then when the request is granted, the requestor has shared access to that node and implicitly to each descendant of that node. These two access modes lock an entire sub-tree rooted at the requested node.

Our goal is to find some technique for implicitly locking an entire sub-tree. In order to lock a sub-tree rooted at node R in share or exclusive mode it is important to prevent locks on the ancestors of R that might implicitly lock R and its descendants in an incompatible mode. Hence a new access mode, intention mode (I), is introduced. Intention mode is used to “tag” (lock) all ancestors of a node to be locked in share or exclusive mode. These tags signal the fact that locking is being done at a "finer" level and thereby prevents implicit or explicit exclusive or share locks on the ancestors.

The protocol to lock a sub-tree rooted at node R in exclusive or share

The protocol to lock a sub-tree rooted at node R in exclusive or share mode is to first lock all ancestors of R in intention mode, and then to lock node R in exclusive or share made. For example, using the figure above, to lock a particular file one should obtain intention access to the database, to the area containing the file, and then request exclusive (or share) access to the file itself. This implicitly locks all records of the file in exclusive (or share) mode.

5.7.6.2. ACCESS MODES AND COMPATIBILITY

We say that two lock requests for the same node by two different transactions are compatible if they can be granted concurrently. The mode of the request determines its compatibility with requests made by other transactions.. The three modes X, S, and I are incompatible with one another, but distinct S requests may be granted together and distinct I requests may be granted together,.

The compatibilities among modes derive from their semantics. Share mode allows reading but not modification of the corresponding resource by the requestor and by other transactions. The semantics of exclusive mode is that the grantee may read and modify the resource, but no other transaction may read or modify the resource while the exclusive lock is set. The reason for dichotomizing share and exclusive access is that several share requests can be granted concurrently (are compatible) whereas an exclusive request is not compatible with any other request. Intention mode was introduced to be incompatible with share and exclusive mode (to prevent share and exclusive locks). However, intention mode is compatible with itself since two transactions having intention access to a node will explicitly lock descendants of the node in X, S or I mode and thereby will either be compatible with one another or will be scheduled on the basis of their requests at the finer level. For example, two transactions can simultaneously be granted the database and some area and some file in intention mode. In this case their explicit locks on particular records in the file will resolve any conflicts among them.

The notion of intention mode is refined to intention share mode (IS) and intention exclusive (IX) for two reasons: intention share mode only requests share or intention share locks at the lower nodes of the tree (i.e. never requests an exclusive lock below the intention share node.) Hence IS mode is compatible with S mode. Since read only is a common form of access it will be profitable to distinguish this for greater concurrency. Secondly, if a transaction has an intention share lock on a node it can convert this to a share lock at a later time, but one cannot convert an intention exclusive lock to a share lock on a node, Rather to get the combined rights of share node and intention exclusive mode one must obtain an X or SIX mode lock. (This issue is discussed in the section on rerequests below).

We recognize one further refinement of modes, namely share and intention exclusive mode (SIX). Suppose one transaction wants to read an entire sub-tree and to update particular nodes of that sub-tree. Using the modes provided so far it would have the options of: (a) requesting exclusive access to the root of the sub-tree and doing no further locking or (b) requesting intention exclusive access to the root of the sub-tree and explicitly locking the lover nodes in intention; share or exclusive mode. Alternative (a) has low concurrency. If only a small fraction of the read nodes are updated then alternative (b) has nigh locking overhead, The correct access mode would be share access to the sub-tree thereby allowing the transaction to read all nodes of the sub-tree without further locking and intention exclusive access to the sub-tree thereby allowing the transaction co set exclusive locks on those nodes in the sub-tree that are to be updated and IX or SIX locks on the intervening nodes. SIX mode is introduced because this is a common case. It is compatible with IS mode since other transactions requesting IS mode will explicitly lock lower nodes in IS or S mode thereby avoiding any updates (IX or X mode) produced by the SIX mode transaction. However SIX mode is not compatible with IX, S, SIX or X mode requests.

The table below gives the compatibility of the request modes, where null mode (NL) represents the absence of a request.

	NL	IS	IX	S	SIX	X
IS	YES	YES	YES	YES	YES	NO
IX	YES	YES	YES	NO	NO	NO
S	YES	YES	NO	NO	NO	NO
SIX	YES	YES	NO	NO	NO	NO
X	YES	NO	NO	NO	NO	NO

Table 1. Compatibilities among access modes.

To summarize, we recognize six modes of access to a resource:

NL: Gives no access to a node, (i.e. represents the absence of a request of a resource.)

IS: Gives intention share access to the requested node and allows the requestor to lock descendant nodes in S or IS mode. (It does no implicit locking.)

IX: Gives intention exclusive access to the requested node and allows the requestor to explicitly lock descendants in X, S, SIX, IX or IS mode. (It does no implicit locking.)

S: Gives shared access to the requested node and to all descendants of the requested node without setting further locks. (It implicitly sets S locks on all descendants of the requested node.)

SIX: Gives share and intention exclusive access to the requested node. In particular, it implicitly locks all descendants of the node in share mode and allows the requestor to explicitly lock descendant nodes in X, SIX or IX mode.)

X: Gives exclusive access to the requested node and to all descendants of the requested node without setting further locks. (It implicitly sets X Locks on all descendants. Locking lover nodes in S or IS mode would give no increased access.)

IS mode is the weakest non-null form of access to a resource. It carries fever privileges than IX or S modes. IX mode allows IS, IX, S, SIX and X mode locks to be set on descendant nodes while S mode allows read-only access to all descendants of the node without further locking. SIX mode carries the privileges of S and of IX mode (hence the name SIX). X mode is the most privileged form of access and allows reading and writing of all descendants of a node without further locking. Hence the modes can be ranked in the partial order of privileges shown the figure below. Note that it is not a total order since IX and S are incomparable.

```
       X
       |
      SIX
       |
    +--+--+
    |     |
    S    IX
    |     |
    +--+--+
       |
      IS
       |
      NL
```

Figure 2. The partial ordering of modes by their privileges.

5.7.6.3. RULES FOR REQUESTING NODES

The implicit locking of nodes will not work if transactions are allowed to leap into the middle of the tree and begin locking nodes at random. The implicit locking implied by the S and X modes depends on all transactions obeying the following protocol:

(a) Before requesting an S or IS lock on a node, all ancestor nodes of the requested node must be held in IX or IS mode by the requestor.

(b) Before requesting an X, SIX or IX lock on a node, all ancestor nodes of the requested node must be held in SIX or IX mode by the requestor.

(c) Locks should be released either at the end of the transaction (in any order) or in leaf to root order. In particular, if locks are not held to end of transaction, one should not hold a lock after releasing its ancestors.

To paraphrase this, locks are requested root to leaf, and released leaf to root. Notice that leaf nodes are never requested in intention mode since they have no descendants, and that once a node is acquired in S or X mode, no further explicit locking is required at lover levels.

5.7.6.4. SEVERAL EXAMPLES

To lock record R for read:

```
lock database             with mode =IS
lock area containing R    with mode =IS
lock file containing R    with mode =IS
lock record R             with mode = S
```

Don't panic, the transaction probably already has the database, area and file lock.

To lock record R for write-exclusive access:

```
lock database             with mode =IX
lock area containing R    with mode =IX
lock file containing R    with mode =IX
lock record R             with mode = X
```

Note that if the records of this and the previous example are distinct, each request can be granted simultaneously to different transactions even though both refer to the same file.

To lock a file F for read and write access:

```
lock database           with mode =IX
lock area containing F  with mode =IX
lock file P             with mode = X
```

Since this reserves exclusive access to the file, if this request uses the same file as the previous two examples it or the other transactions will have to wait. Unlike examples 1, 2 and 4, no additional locking need be done (at the record level).

To lock a file F for complete scan and occasional update:

```
lock database           with node =IX
lock area containing F  with mode =IX
lock file F             with mode =SIX
```

Thereafter, particular records in F can be locked for update by locking the desired records in X mode. Notice that (unlike the previous example) this transaction is compatible with the first example. This is the reason for introducing SIX mode.

To quiesce the data base:

```
lock data base          with mode =X.
```

Note that this locks everyone else out.

5.7.6.5. DIRECTED ACYCLIC GRAPHS OF LOCKS

The notions so far introduced can be generalized to work for directed acyclic graphs (DAGs) of resources rather than simply hierarchies of resources. A tree is a simple DAG. The key observation is that to implicitly or explicitly lock a node, one should lock all the parents of the node in the DAG, and so by induction lock all ancestors of the node. In particular, to lock a subgraph one must implicitly or explicitly lock all ancestors of the subgraph in the appropriate mode (for a tree there is only one parent). To give an example of a non-hierarchical structure, imagine the locks are organized as:

```
    DATA BASE
        |
      AREAS
        |
  +-----+-----+
  |           |
FILES      INDICES
  +-----+-----+
        |
     RECORDS
```

Figure 3. A non-hierarchical lock graph.

We postulate that areas are "physical” notions and that files, indices and records are logical notions. The database is a collection of areas. Each area is a collection of files and indices. Each file has a few corresponding indices in the same area. Each record belongs to some file and to its corresponding indices. A record is comprised of field values and some field is indexed by the index associated with the file containing the record. The file gives a sequential access path to the records and the index gives an associative access path to the records based on field values. Since individual fields are never locked, they do not appear in the lock graph.

To write a record A in file P with index I:

```
lock data base          with mode =IX
lock area containing F  with mode =IX
lock file F             with mode =IX
lock index I            with mode =IX
lock record R           with mode = X
```

Note that all paths to record R are locked. Alternatively, one could lock F and I in exclusive mode thereby implicitly locking R in exclusive mode.

To give a more complete explanation we observe that a node can be locked explicitly (by requesting it) or implicitly (by appropriate explicit locks on the ancestors of the node) in one of five modes: IS, IX. S, SIX, X. However, the definition of implicit locks and the protocols for setting explicit locks have to be extended for DAG's as follows:

A node is implicitly granted in S mode to a transaction if at least one of its parents is (implicitly or explicitly) granted to the transaction in S, SIX or X mode. By induction, that means that at least one of the node's ancestors must be explicitly granted in S, SIX, or X mode to the transaction.

A node is implicitly granted in X mode if all of its parents are (implicitly or explicitly) granted to the transaction in X mode. By induction, this is equivalent to the condition that all nodes in some cut set of the collection of all paths leading from the node to the roots of the graph are explicitly granted to the transaction in X mode and all ancestors of nodes in the cut set are explicitly granted in IX or SIX mode.

By examination of the partial order of modes (see figure above), a node is implicitly granted in IS mode if it is implicitly granted in S mode, and a node is implicitly granted in IS, IX, S and SIX mode if it is implicitly granted in X mode.

5.7.6.6. PROTOCOL FOR REQUESTING LOCKS ON A DAG

(a) Before requesting an S or IS lock on a node, one should request at least one parent (and by induction a path to a root) in IS (or greater) mode. As a consequence none of the ancestors along this path can be granted to another transaction in a mode incompatible with IS.

(b) Before requesting IX, SIX or x mode access to a node, one should request all parents of the node in IX (or greater) mode. As a consequence all ancestors will be held in IX (or greater mode) and cannot be held by other transactions in a mode incompatible with IX (i.e. S, SIX, X).

(c) Locks should be released either at the end of the transaction (in any order) or in leaf to root order. In particular, if locks are not held to the end of transaction, one should not hold a lower lock after releasing its ancestors.

To give an example using the non-hierarchical lock graph in the figure above, a sequential scan of all records in file F need not use an index so one can get an implicit share lock on each record in the file by:

```
lock data base          with mode =IS
lock area containing F  with mode =IS
lock file F             with mode = S
```

This gives implicit S mode access to all records in P. Conversely, to read a record in a file via the index I for file P, one need not get an implicit or explicit lock on file F:

```
        lock data base           with mode =IS
        lock area containing R  with mode =IS
        lock index I             with mode = S
```

This again gives implicit S mode access to all records in index I (in file F). In both these cases, only one path was locked for reading.

But to insert, delete or update a record R in file F with index I one must get an implicit or explicit lock on all ancestors of R.

The first example of this section showed how an explicit X lock on a record is obtained. To get an implicit X lock on all records in a file one can simply lock the index and file in X mode, or lock the area in X mode. The latter examples allow bulk load or update of a file without further locking since all records in the file are implicitly granted in X mode.

5.7.6.7. PROOF OF EQUIVALENCE OF THE LOCK PROTOCOL

We will now prove that the described lock protocol is equivalent to a conventional one which uses only two modes (S and X), and which explicitly locks atomic resources (the leaves of a tree or sinks of a DAG).

Let G =(N, A) be a finite (directed acyclic) graph where N is the set of nodes and A is the set of arcs. G is assumed to be without circuits (i.e. there is no non-null path leading from a node n to itself). A node p is a parent of a node n, and n is a child of p if there is an arc from p to n. A node n is a source (sink), if n has no parents (no children). An ancestor of node n is any node (including n) in a path from a source to n. A node-slice of a sink n is a collection of nodes such that each path from a source to n contains at least one node of the slice. Let Q be the set of all sinks of G.

We also introduce the set pf lock modes M = {NL, IS, IX, S, SIX, X} and the compatibility matrix C : AxM-> (YES, NO) described in Table 1. Let c : AxM -> {YES, NO} be the restriction of C to m ={NL, S, X}.

A lock graph is a mapping L : N->M such that:

(a) If L(n) is in {IS, S} then either n is a source or there exists a parent p of n such that L(p) is in { IS, IX, S, SIX, X). By induction there exists a path from a source to n such that L(ni) takes only values in {IS, IX, S, SIX, X} on all nodes ni of that path. Equivalently, L(ni) is not equal to NL on the path.

(b) If L(n) is in {IX, SIX, X} then either n is a root, or for all parents p1,..., pk of n we have L(pi) is in { IX. SIX, X} for i = 1,..., k . By induction L takes only values in {IX, SIX, X} on all the ancestors of n.

The interpretation of a lock-graph is that it gives a map of the explicit locks held by a particular transaction observing the six state lock protocol described above. The notion of projection of a lock-graph is now introduced to model the set of implicit locks on atomic resources acquired by a transaction.

The projection of a lock-graph L is the mapping p: Q->m (from sinks to modes) constructed as follows:

(a) p(n)=X if there exists a node-slice {n1,.. , ns} of N such that p(ni)=X for each node in the slice.

(b) p(n)=S if (a) is not satisfied and there exists an ancestor na of N such that p(na) is in {S, SIX, X}.

(c) p(n)=NL if (a) and (b) are not satisfied.

Two lock-graphs L1 and L2 are said to be compatible if C(L1(n), L2(n)) = YES for all n in N. Similarly two projections p1 and p2 are compatible if c(p1(n), p2(n)) = YES for all sink nodes n in Q.

Theorem: If two lock-graphs L1 and L2 are compatible, then their projections P1 and P2 are compatible. In other words if the explicit locks set by two transactions do not conflict then also the three-state locks implicitly acquired do not conflict.

Proof*:* Assume that L1 and L2 are incompatible. We want to prove that P1 and P2 are incompatible. By definition of compatibility there must exist a sink n such that L1(n)=X and L2(n) is in {S, X} (or vice versa). By definition of projection, there must exist a node-slice {n1,..., ns} of N such that L1(n1)=...=L1(ns)=X. Also there must exist an ancestor na of n such that L2(na) is in {S, SIX, X}. From the definition of lock-graph there is a path Path1 from a source to na on which L2 does not take the value NL. If Path1 intersects the node-slice at ni then L1 and L2 are incompatible since L1(ni)=X which is incompatible with the non-null value of L2(ni). Hence the theorem is proved.

Alternatively there is a path Path2 from na to the sink n that intersects the node-slice at ni. From the definition of lock-graph L1 takes a value in {IX, SIX, X} on all ancestors of ni. In particular L1(na) is in {IX, SIX, X}. Since L2(na) is in {S, SIX, X} we have C(L1(na), L2 (na))=NO.

Q. E. D.

5.7.7. LOCK MANAGEMENT PRAGMATICS

Thus far we have discussed when to lock (lock before access and hold locks to commit point) and why to lock (to guarantee consistency and to make recovery possible without cascading transaction backup,) and what to lock (lock at a granularity that balances concurrency against instruction overhead in setting locks.) The remainder of this section will discuss issues associated with how to implement a lock manager.

5.7.7.1. THE LOCK MANAGER INTERFACE

This is a simple version of the System R lock manager.

5.7.7.1.1. LOCK ACTIONS

Lock manager has two basic calls:

```
LOCK <lock>, <mode>, <class>, <control>
```

where <lock> is the resource name (in System R for example an eight byte name). <mode> is one of the lock modes { S | X | SIX | IX | IS }. <class> is a notion described below. <control> can be either WAIT in which case the call is synchronous and waits until the request is granted or is cancelled by the deadlock detector, or <control> can be TEST in which case the request is canceled if it cannot be granted immediately.

```
UNLOCK <lock>, <class>
```

releases the specified lock in the specified class. If the <lock> is not specified, all locks held in the specified class are released.

5.7.7.1.1.2. LOCK NAMES

The association between lock names and objects is purely a convention. Lock manager associates no semantics with names. Generally the first byte is reserved for the subsystem (component) identifier and the remaining seven bytes name the object.

For example, data manager might use bytes (2... 4) for the file name and bytes (5... 7) for the record name in constructing names for record locks.

Since there are so many locks, one only allocates those with non-null queue headers. (i.e. free locks occupy no space.) Setting 1 lock consists of hashing the lock name into a table. If the header already exists, the request enqueues on it, otherwise the request allocates the lock header and places it in the hash table. When the queue of a lock becomes empty, the header is deallocated (by the unlock operation).

5.7.7.1.1.2. LOCK CLASSES

Many operations acquire a set of locks. If the operation is successful, the locks should be retained. If the operation is unsuccessful, or when the operation commits, the locks should be released. In order to avoid double bookkeeping, the lock manager allows users to name sets of locks (in the new DBTG proposal these are called keep lists, in IMS program isolation these are called *Q class locks).

For each lock held by each process, lock manager keeps a list of <class, count> pairs. Each lock request for a class increments the count for that class. Each unlock request decrements the count. When all counts for all the lock's classes are zero then the lock is not held by the process.

5.7.7.1.4. LATCHES

Lock manager needs a serialization mechanism to perform its function (e.g. inserting elements in a queue or hash chain). It does this by implementing a lower level primitive called a latch. Latches are semaphores. They provide a cheap serialization mechanism without providing the expensive features like deadlock detection, class tracking, and modes of sharing (beyond S or X). They are used by lock manager and by other performance critical managers (notably buffer manager and log manager).

5.7.7.1.5. Performance of Lock Manager

Lock manager is about 3300 lines of (PL/I like) source code. It depends critically on the Compare and Swap logic provided by the multiprocessor feature of System 370. It comprises three percent of the code and about ten percent of the instruction execution of a program in System R (this may vary a great deal.) A lock-unlock pair currently costs 350 instructions but if these notes are ever finished, this will be reduced to 120 instructions (this should reduce its slice of the execution pie.) A latch-unlatch pair requires 10 instructions (they expand in-line). (Initially they required 120 instructions but a careful redesign improved this dramatically.)

5.7.7.2. SCHEDULING AND GRANTING REQUESTS

Thus far we have described the semantics of the various request modes and have described the protocol that requestors must follow. To complete the discussion we discuss how requests are scheduled and granted.

The set of all requests for a particular resource are kept in a queue sorted by some fair scheduler. By "fair" we mean that no particular transaction will be delayed indefinitely. First-in first-out is the simplest fair scheduler and we adopt such a scheduler for this discussion modulo deadlock preemption decisions.

The group of mutually compatible requests for a resource appearing at the head of the queue is called the granted group. All these requests can be granted concurrently. Assuming that each transaction has at most one request in the queue then the compatibility of two requests by different transactions depends only on the modes of the requests and may be computed using Table 1. Associated with the granted group is a group mode is the supremum mode of the members of the group which is computed using Figure 2 or Table 3. Table 2 gives a list of the possible types of requests that can coexist in a group and the corresponding mode of the group.

Table 2. Possible request groups and their group mode. Set brackets indicate that several such requests may be present.

Request Modes	Group Mode
X	X
{ SIX, {IS} }	SIX
{ S, {S}, {IS} }	S
{ IX, {IX}, {IS} }	IX
{ IS, {IS} }	IS

The figure below depicts the queue for a particular resource, showing the requests and their modes. The granted group consists of five requests and has group mode IX. The next request in the queue is for S mode that is incompatible with the group mode IX and hence must wait.

```
**********************************
*   GRANTED GROUP: GROUPMODE =IX *
* |IS|-|IX|-|IS|--|IS|-|IS|-|IS|-*-|S|-|IS|-|X|-|IS|-|IX|
**********************************
```

Figure 5. The queue of requests for a resource.

When a new request for a resource arrives, the scheduler appends it to the end of the queue. There are two cases to consider: either someone is already waiting, or all outstanding requests for this resource are granted (i.e. no one is waiting). If waiters exist, then the request cannot be granted and the new request must wait. If no one is waiting and the new request is compatible with the granted group mode then the new request can be granted immediately. Otherwise the new request must wait its turn in the queue, and in the case of deadlock it may preempt some incompatible requests in the queue. (Alternatively the new request could be canceled. In Figure 5 all the requests decided to wait.)

.

When a particular request leaves the granted group, the mode of the group may change. If the mode of the first waiting request is compatible with the new mode of the granted group, then the waiting request is granted. In Figure 5 if the IX request leaves the group, then the group mode becomes IS which is compatible with S and so the S request may be granted. The new group mode will be S and since this is compatible with the IS mode. The IS requests following the S request may also join the granted group. This produces the situation depicted in Figure 6.

```
*********************************
*   GRANTED GROUP: GROUPMODE = S *
* |IS|-|IS|-|IS|--|IS|-|S|-|IS|--*-|X|-|IS|-|IX|
*********************************
```

Figure 6. The queue after the IX request is released.

The X request of Figure 6 will not be granted until all requests leave the granted group since it is not compatible with any of them.

5.7.7.3. CONVERSIONS

A transaction might re-request the same resource for several reasons: Perhaps it has forgotten that it already has access to the record; after all, if it is setting many locks it may be simpler to just always request access to the record rather than first asking itself "have I seen this record before". The lock manager has all the information to answer this question and it seems wasteful to duplicate. Alternatively, the transaction may know it has access to the record, but wants to increase its access mode (for example from S to X mode if it is in a read, test, and sometimes update scan of a file). So the lock manager must be prepared for re-requests by a transaction for a lock. We call such re-requests conversions.

When a request is found to be a conversion, the old (granted: node of the requestor to the resource and the newly requested mode are compared using Table 3 to compute the new mode which is the supremum of the old and the requested mode (see Figure 2.)

Table 3. The new mode given the requested and old mode.

	IS	IX	S	SIX	X
IS	IS	IX	S	SIX	X
IX	IX	IX	SIX	SIX	X
S	S	SIX	S	SIX	X
SIX	SIX	SIX	SIX	SIX	X
X	X	X	X	X	X

So for example, if one has Ix mote and requests S mode then the new mode is SIX.

If the new mode is equal to the old mode (note it is never less than the old mode) then the request can be granted immediately, and the granted mode is unchanged. If the new mode is compatible with the group mode of the other members of the granted group (a requestor is always compatible with himself), then again the request can be granted immediately. The granted mode is the new mode and the group mode is recomputed using Table 2. In all other cases, the requested conversion must wait until the group mode of the other granted requests is compatible with the new mode. Note that this immediate granting of conversions over waiting requests is a minor violation of fair scheduling.

If two conversions are waiting, each of which is incompatible with an already granted request of the other transaction, then a deadlock exists and the already granted access of one must be preempted. Otherwise there is a way of scheduling the waiting conversions: namely, grant a conversion when it is compatible with all other granted nodes in the granted group. (Since there is no deadlock cycle, this is always possible.)

The following example may help to clarify these points. Suppose the queue for a particular resource is:

```
****************************
*   GROUPMODE =IS          *
*   | IS|---|IS|-----------------------
****************************
```

Figure 7. A simple queue.

Now suppose the first transaction wants to convert to X mode. It must wait for the second (already granted)request to leave the queue. If it decides to wait then the situation becomes:

```
****************************
* GROUPMODE = IS           *
* |IS<-X|---|IS|----------------------
****************************
```

Figure 8. A conversion to X mode waits.

No new request may enter the granted group since there is now a conversion request waiting. In general, conversions are scheduled before new requests. If the second transaction now converts to IX, SIX, or S mode it may be granted immediately since this does not conflict with the granted (IS) mode of the first transaction. When the second transaction eventually leaves the queue, the first conversion can be made:

```
****************************
* GROUPMODE = X           *
* |X|--------------------------------
****************************
```

Figure 9. One transaction leaves and the conversion is granted. However, if the second transaction tries to convert to exclusive mode one obtains the queue:

```
****************************
* GROUPMODE = IS           *
* |IS<-X|---|IS<-X|----------------------
****************************
```

Figure 10. Two conflicting conversions are waiting.

Since X is incompatible with IS (see Table 1). this situation implies that each transaction is waiting for the other to leave the queue (i.e. deadlock) and so one transaction must be preempted. In all other cases (i.e. when no cycle exists) there is a way to schedule the conversions so that no already granted access is violated.

5.7.7.4. DEADLOCK DETECTION

One issue the lock manager must deal with is deadlock. Deadlock consists of each member of a set of transactions waiting for some other member of the set to give up a lock. Standard lore has it that one can have timeout, or deadlock-prevention, or deadlock detection.

Timeout causes waits to be denied after some specified interval. It has the property that as the system becomes more congested, more and more transactions time out (because time runs slower and because more resources are in use so that one waits more). Also timeout puts an upper limit on the duration of a transaction. 1n general the dynamic properties of timeout make it acceptable for a lightly loaded system but inappropriate for a congested system.

Deadlock prevention is achieved by: requesting all locks at once, or requesting locks in a specified order, or never waiting for a lock, or . . . In general deadlock prevention is a bad deal because one rarely knows what locks are needed in advance (consider looking something up in an index,) and consequently, one locks too much in advance.

Although some situations allow deadlock prevention, general systems tend to require deadlock detection. IMS, for example, started with a deadlock prevention scheme (intent scheduling) but was forced to introduce a deadlock detection scheme to increase concurrency (Program Isolation).

Deadlock detection and resolution is no big deal in a data management system environment. The system already has lots of facilities for transaction backup so that it can deal with other sorts of errors. Deadlock simply becomes another (hopefully infrequent) source of backup. As will be seen, the algorithms for detecting and resolving deadlock are not complicated or time consuming.

The deadlock detection-resolution scenario is:

- Detect a deadlock.
- Pick a victim (a lock to preempt from a process.)
- Back out a victim that will release lock.
- Grant a waiter.
- (optionally) Restart victim.

Lock manager is only responsible for deadlock detection and victim selection, Recovery management implements transaction backup and controls restart logic.

5.7.7.4.1. HOW TO DETECT DEADLOCK

There are many heuristic ways of detecting deadlock (e. g. linearly order resources or processes and declare deadlock if ordering is violated by a wait request.)

Here we restrict ourselves to algorithmic solutions.

The detection of deadlock may be cast in graph-theoretic terms. We introduce the notion of the wait-for graph.

- . The nodes of the graph are transactions and locks.
- . The edges of the graph are directed and are constructed as follows:
    - If lock L is granted to transaction T then draw an edge from L to T.
    - If transaction T is waiting for transaction L then draw an edge from T to L.

At any instant, there is a deadlock if and only if the wait-for graph has a cycle. Hence deadlock detection becomes an issue of building the wait-for graph and searching it for cycles.

Often this “transaction waits for lock waits for transaction” graph can be reduced to a smaller “transaction waits for transaction” graph. The larger graph need be maintained only if the identities of the locks in the cycle are relevant. I know of no case where this is required.

5.7.7.4.2. WHEN TO LOOK FOR DEADLOCK

One could opt to look for deadlock:

- Whenever anyone waits.
- Periodically.
- Never.

One could look for deadlock continuously. Releasing a lock or being granted a lock never creates a deadlock. So one should never look for deadlock more frequently than when a wait occurs.

The cost of looking for deadlock every time anyone waits is:

- Continual maintenance of the wait-for graph.
- Almost certain failure since deadlock is (should be) rare (i.e. wasted instructions).

The cost of periodic deadlock detection is:

- Detecting deadlocks late.

By increasing the period one decreases the cost of deadlock and the probability of successfully finding one. For each situation there should be an optimal detection period.

```
  | *        +    * cost of detection
C |   *    +      + cost of detecting late
O |     * +
S |      +*
T |     +   *
  |    +     *
  |   +        *
  +-------|-------------
         optimal
         PERIOD ->
```

Never testing for deadlocks is much like periodic deadlock detection with a very long period. All systems have a mechanism to detect dead programs (infinite loops, wait for lost interrupt,...) This is usually a part of allocation and resource scheduling. It is probably outside and above deadlock detection.

Similarly, if deadlock is very frequent, the system is thrashing and the transaction scheduler should stop scheduling new work, and perhaps abort some current work to reduce this thrashing. Otherwise the system is likely to spend the majority of its time backing up.

<u>5.7.7.4.3. WHAT TO DO WHEN DEADLOCK IS DETECTED.</u>

All transactions in a deadlock are waiting. The only way to get things going again is to grant some waiter. But, this can only be achieved after a lock is preempted from some holder. Since the victim is waiting, he will get the 'deadlock' response from lock manager rather than the 'granted' response.

In breaking the deadlock some set of victims will be preempted. We want to minimize the amount of work lost by these preemptions. Therefore, deadlock resolution wants to pick a minimum cost set of victims to break deadlocks.

Transaction management must associate a cost with each transaction. In the absence of policy decisions: the cost of a victim is the cost of undoing his work and then redoing it. The length of the transaction log is a crude estimate of this cost. At any rate, transaction management must provide lock management with an estimate of the cost of each transaction. Lock manager may implement either of the following two protocols:

- For each cycle, choose the minimum cost victim in that cycle.
- Choose the minimum cost cut-set of the deadlock graph.

The difference between these two options is best visualized by the picture:

```
+--->L1 ---->+<----L2 <---+
|            |            |
|            V            |
|            |            |
T1           T2           T3
^            |            ^
|            |            |
|            |            |
+<-----------+----------->+
```

If T1 and T3 have a cost of 2 and T2 has a cost of 3 then a cycle-at-a-time algorithm will choose T1 and T3 as victims; whereas, a minimal cut set algorithm will choose T2 as a victim.

The cost of finding a minimal cut set is considerably greater (seems to be NP complete) than the cycle-at-a-time scheme. If there are N common cycles, the cycle-at-a-tine scheme is at most N times worse than the minimal cut set scheme. So it seems that the cycle-at-a-time scheme is better.

5.7.7.4.4. LOCK MANAGEMENT IN A DISTRIBUTED SYSTEM.

To repeat the discussion in the section OIL distributed transaction management, if a transaction wants to do work at a new node, some process of the transaction must request that the node construct a cohort and that the cohort go into session with the requesting process (see section on data communications for a discussion of sessions.) The picture below shows this.

```
NODE1
+----------+
| ******** |
| * T1P2 * |
| ****#*** |
+-----#----+
      #
 =========
| SESSION1 |
 =========
      #
NODE2 #
+-----#----+
| ****#*** |
| * T1P6 * |
| ******** |
+----------+
```

A cohort carries both the transaction name T1 and the process name (in NODE1 the cohort of T1 is process P2 and in NODE2 the cohort of T1 is process P6.)

The two processes can nor converse and carry out the work of the transaction. If one process aborts, they should both abort, and if one process commits they should both commit.

The lock manager of each node can keep its lock tables in any form it desires. Further, deadlock detectors running in each node may use any technique they like to detect deadlocks among transactions that run exclusively in that node. We call such deadlocks local deadlocks. However, just because there are no cycles in the local wait-for graph, does not mean that there are no cycles. Gluing acyclic local graphs together might produce a graph with cycles (See the example bellow.) So, the deadlock detectors of each node will have to agree on a common protocol in order to handle deadlocks involving distributed transactions. We call such deadlocks global deadlocks.

Inspection of the following figure may help to understand the nature of global deadlocks. Note that transaction T1 has two processes Pl and P2 in nodes 1 and 2 respectively. Pl is session-waiting for its cohort P2 to do some work. P2, in the process of doing this work, needed access to FILE2 in NODE2. But FILE2 is locked exclusive by another process (P4 of NODE2) so P2 is in lock wait state. Thus the transaction T1 is waiting for FILE2. Now Transaction T2 is in a similar state, one of its cohorts is session waiting for the other which in turn is lock waiting for FILE1. In fact transaction T1 is waiting for FILE2, which is granted to transaction T2 that is waiting for file FILE1 that is granted to transaction T1. A global deadlock if you ever saw one.

```
NODE1
+-----------------------------------------------+
|            lock              lock             |
| ********   grant  *********  wait  ********   |
| * T1P1 * <------- * FILE1 * <----- *T2P3  *   |
| ********          *********        ********   |
|     #                                   #     |
|     # session                           #     |
|     # wait                              #     |
+-----#-----------------------------------#-----+
      #                                   #
      v                                   #
 ==========                          ==========
 |SESSION1 |                        | SESSION2 |
 ==========                          ==========
      #                                   #
NODE2 #                                   #
+-----#-----------------------------------#-----+
|     #                          session  #     |
|     #                             wait  #     |
|     #                                   #     |
|     #      lock              lock      #     |
| ********   wait   *********  grant ********   |
| * T1P2 * -------> * FILE2 * -----> *T2P3  *   |
| ********          *********        ********   |
+-----------------------------------------------+
```

The notion of wait-for graph must be generalized to handle global deadlock. The nodes of the graph are processes and resources (sessions are resources). The edges of the graph are constructed as follows:

o Draw a directed edge free a process to a resource if

  - the process is in lock wait for the resource,

  - or the process is in session-wait for the resource (session)-

o Draw a directed edge from a resource to a process if

  - the resource is lock granted to the process

  - or it is a session of the process and the process is not in session-wait on it.

A local deadlock is a

lock wait ->. . . .-> lockwait   cycle.

A global deadlock is a

lockwait->... -> sessionwait -> lockwait ->...-> sessionwait   cycle

5.7.7.5.1. HOW TO FIND GLOBAL DEADLOCKS

The finding of local deadlocks has already been described. To find global deadlocks, a distinguish task, called the global deadlock detector is started in some distinguished node. This task is in session with all local deadlock detectors and coordinates the activities of the local deadlock detectors. This global deadlock detector can run in any node, but probably should be located to minimize its communication distance to the lock managers.

Each local deadlock Detector needs to find all potential global deadlock paths in his node. In the previous section it was shown that a global deadlock cycle has the form:

lockwait ->... -> sessionwait ->lockwait ->...-> sessionwait ->

So each local deadlock detector periodically enumerates all

session -> lockwait ->...-> sessionwait

paths in his node by working backwards from processes that are in sessionwait (as opposed to console wait, disk wait, processor wait,…) Starting at such a process it sees if some local process is lock waiting for this process. If so the deadlock detector searches backwards looking for some process which has a session in progress.

When such a path is found, the following information is sent to the global deadlock detector:

- Sessions and transactions at endpoints of the path and their local preemption costs.
- The minimum cost transaction in the path and his local pre-emption cost.

(It may make sense to batch this information to the global detector.) Periodically, the global deadlock detector:

- collects these messages,
- glues all these paths together by matching up sessions, and
- enumerates cycles and selects victims just as in the local deadlock detector case.

One tricky point is that the cost of a distributed transaction is the sum of the costs of its cohorts. The global deadlock detector approximates this cost by summing the costs of the cohorts of the transaction known to it (not all cohorts of a deadlocked transaction will be in known to the global deadlock detector.)

When a victim is selected, the lock manager of the node the victim is waiting in is informed of the deadlock. The local lock manager in turn informs the victim with a deadlock return.

The use of periodic deadlock detection (as opposed to detection every time anyone waits)is even more important for a distributed system than for a centralized system, The cost of detection is much higher in a distributed system. This will alter the intersection of the cost of detection and cost of detecting late curves.

If the network is really large the deadlock detector can be staged. That is, we can look for deadlock among four nodes, then among sixteen nodes, and so on.

If one node crashes, then its partition of the system is unavailable.

In this case, its cohorts in other nodes can wait for it to recover or they can abort. If the down node happens to house the global lock manager then no global deadlocks will be detected until the node recovers. If this is uncool, then the lock managers can nominate a new global lock manager whenever the current one crashes. The new manager can run in any node that can be in session with all other nodes. The new global lock manager collects the local graphs and goes about gluing them together, finding cycles, and picking victims.

5.7.7.6. RELATIONSHIP A OPERATING SYSTEM LOCK MANAGER

Most operating systems provide a lock manager to regulate access to files and other system resources. This lock manager usually supports a limited set of lock names, the modes: share, exclusive and beware, and has some form of deadlock detection. These lock managers are usually

net prepared for the demands of a data management system (fast calls, lots of locks, many modes, lock classes,.,.)The basic lock manager could be extended and refined and in time that is what will happen.

There is a big problem about having two lock managers in the same host. Each may think it has no deadlock but if their graphs are glued together a “global” deadlock exists. This makes it very difficult to build on top of the basic lock manager.

5.7.7.7. THE CONVOY PHENOMENON: PREEMPTIVE SCHEDULING IS BAD

Lock manager has strong interactions with the scheduler. Suppose that there are certain high traffic shared system resources. Operating on these resources consists of locking them, altering them and then unlocking them (the buffer pool and log are examples of this.) These operations are designed to be very fast so that the resource is almost always free. In particular the resource is never held during an I/O operation, For example, the buffer manager latch is acquired every 1000 instructions and is held for about 50 instructions.

If the system has no preemptive scheduling then on a uni-processor, then when a process begins the resource is free and when he completes the resource is free (because he does not hold it when he does I/O or yields the processor.) On a multi-processor, if the resource is busy, the process can sit in a busy wait until the resource is free because the resource is known to be held by others for only a short time.

If the basic system has a preemptive scheduler, and if that scheduler preempts a process holding a critical resource (e. g. the log latch) then terrible things happen: All other processes waiting for the latch will dispatched before this process is dispatched, and because the resource is high traffic each of these processes requests and waits for the resource. Ultimately the holder of the resource is re-dispatched and he almost immediately grants the latch to the next waiter. But because it is high traffic, the process almost immediately rerequests the latch (i.e. about 1000 instructions later.) Fair scheduling requires that he wait, so he goes on the end of the queue waiting for those ahead of him. This queue of waiters is called a convoy. It is a stable phenomenon: once a convoy is established it persists for a very long time.

We (System R) have found several solutions to this problem. The obvious solution is to eliminate such resources. That is a good idea, and can be achieved to some degree by refining the granularity of the lockable unit (e-g, twenty buffer manager latches rather than just one.) However, if a convoy ever forms on any of these latches it will be stable so that is not a solution. I leave it as an exercise for the reader to find a better solution to the problem.

Engles, “Currency and Concurrency in the COBOL Data Base Facility”, in *Modeling in Data Base Management Systems*, Nijssen editor, North Holland, 1976 (A nice discussion of how locks are used.)

Eswaran et. al. "On the Notions of Consistency and Predicate Locks in a Relational Database System." CACM, Vol. 19, No. 11, November 1976. (Introduces the notion of consistency, ignore the stuff on predicate locks.)

"Granularity of Locks and Degrees of Consistency in a Shared Data Base", in *Modeling in Data Base Management Systems*, Nijssen editor, North Holland, 1976 (This section is a condensation and then elaboration of this paper. Hence Franco Putzolu and Irv Traiger should be considered co-authors of this section.)

5.8. RECOVERY MANAGEMENT

5.8.1. MODEL OF ERRORS

In order to design a recovery system, it is important to have a clear notion of what kinds of errors can be expected and what their probabilities are. The model of errors below is inspired by the presentation by Lampson and Sturgis in "Crash Recovery in a Distributed Data Storage System", which may someday appear in the CACM.

We first postulate that all errors are detectable. That is, if r. o one complains about a situation, then it is OK.

5.8.1.1. MODEL OF STORAGE ERRORS

Storage comes in three flavors with independent failure modes and increasing reliability:

- Volatile storage: paging space and main memory,
- On-Line Non-volatile Storage: disks, usually survive crashes. Is more reliable than volatile storage.
- Off-Line Non-volatile Storage: Tape archive. Even more reliable than disks.

To repeat, we assume that these three kinds of storage have independent failure modes.

The storage is blocked into fixed length units called pages that are the unit of allocation and transfer.

Any page transfer can have one of three outcomes:

1. Success (target gets new value)
2. Partial failure (target is a ness)
3. Total failure (target is unchanged)

Any page may spontaneously fail. That is a spec of dust may settle on it or a black hole may pass through it so that it no longer retains its original information,

One can always detect whether a transfer failed or a page spontaneously failed by reading the target page at a later time. (This can be made more and more certain by adding redundancy to the page.)

Lastly, The probability that N "independent" archive pages fail is negligible. Here we choose N=2, (This can be made more and more certain by choosing larger and larger N.)

5.8.1.2. Model of Data Communications Errors

Communication traffic is broken into units called messages via sessions.

The transmission of a message has one of three possible outcomes:

1. Successfully received.
2. Incorrectly received.
3. Not received.

The receiver of the message can detect whether he has received a particular message and whether it gas correctly received.

For each message transmitted, there is a non-zero probability that it will be successfully received.

It is the job of recovery manager to deal with these storage and transmission errors and correct them. This model of errors is implicit in what follows and will appear again in the examples at the end of the section.

5.8.2. OVERVIEW OF RECOVERY MANAGEMENT

A transaction is begun explicitly when a process is allocated or when an existing process issues BEGIN_TRANSACTION. When a transaction is initiated, recovery manager is invoked to allocate the recovery structure necessary to recover the transaction. This process places a capability for the COMMIT, SAVE, and BACKUP calls of recovery manager in the transaction's capability list.

Thereafter, all actions by the transaction on recoverable data are recorded in the recovery log, using log manager. In general, each action performing an update operation should write an undo-log record and a redo-log record in the transaction’s log. The undo log record gives the old value of the object and the redo log record gives the new value (see below).

At a transaction save point, recovery manager records the save point identifier, and enough information so that each component of the system could be backed up to this point.

In the event of a minor error, the transaction may be undone to a save point in which case the application (on its next or pending call) is given feedback indicating that the data base system has amnesia about all recoverable actions since that save point. If the transaction is completely backed-up (aborted), it may or may not be restarted depending on the attributes of the transaction and of its initiating message.

If the transaction completes successfully (commits), then (logically) it is always redone in case of a crash. On the other hand, if it is in-progress at the time of the local or system failure, then the transaction is logically undone (aborted).

Recovery manager must also respond to the following kinds of failures:

- **Action failure**: a particular call cannot complete due to a foreseen condition. In general the action undoes itself (cleans up its component)and then returns to the caller. Examples of this are bad parameters, resource limits, and data not found.

- **Transaction failure**: a particular transaction cannot proceed and so is aborted. The transaction may be reinitiated in some cases. Examples of such errors are deadlock, timeout, protection violation, and transaction-local system errors.
- **System failure**: a serious error is detected below the action interface. The system is stopped and restarted. Errors in critical tables, wild branches by trusted processes, operating system failures and hardware failures are sources of system failure. Most nonvolatile storage is presumed to survive a system failure.
- **Media failure**: a non-recoverable error is detected on some usually reliable (nonvolatile) storage device. The recovery of recoverable data from a media failure is the responsibility of the component that implements it. If the device contained recoverable data the manager must reconstruct the data from an archive copy using the log and then place the result on an alternate device. Media failures do not generally force system failure. Parity error, head crash, dust on magnetic media, and lost tapes are typical media failures. Software errors that make the media unreadable are also regarded as media errors, as are catastrophes such as fire, flood, insurrection, and operator error.

The system periodically makes copies of each recoverable object and keeps these copies in a safe place (archive). In case the object suffers a media error, all transactions with locks outstanding against the object are aborted. A special transaction (a utility) acquires the object in exclusive mode. (This takes the object “off-line".) This transaction merges an accumulation of changes to the object since the object copy was made and a recent archive version of the object to produce the most recent committed version. This accumulation of changes may take two forms: it may be the REDO-log portion of the system log, or it may be a change accumulation log that was constructed from the REDO-log portion of the system log when the system log is compressed. After media recovery, the data is unlocked and made public again.

The process of making an archive copy of an object has many varieties. Certain objects, notably IMS queue space, are recovered from scratch using an infinite redo log. Other objects, notably databases, get copied to some external media that can be used to restore the object to a consistent state if a failure occurs. (The resource may or may not be off-line while the cozy is being made.)

Recovery manager also periodically performs system checkpoint by recording critical parts of the system state in a safe spot in nonvolatile storage (sometimes called the warm start file.)

Recovery manager coordinates the process of system restart system shutdown. In performing system restart, it chooses among:

- **Warm start**: system shut down in controlled manner. Recovery need only locate last checkpoint record and rebuild its control structures.
- **Emergency restart**: system failed in uncontrolled manner. Non-volatile storage contains recent state consistent with the log. However, some transactions were in progress at time of failure and must be redone or undone to obtain most recent consistent state.
- **Cold start:** the system is being brought up with amnesia about prior incarnations. The log is not examined to determine previous state.

5.8.3. RECOVERY PROTOCOLS

All participants in a transaction, including all components understand and obey the following protocols when operating on recoverable objects:

- Consistency lock protocol.
- The DO-UNDO-REDO paradigm for log records.
- Write-Ahead-Log protocol (WAL).
- Two-phase commit protocol.

The consistency lock protocol was discussed in the section on lock management. The remaining protocols are discussed below.

Perhaps the simplest and easiest to implement recovery technique is based on the old-master new-master dichotomy common to most batch data processing systems: If the run fails, one goes back to the old-master and tries again. Unhappily, this technique does not seem to generalize to concurrent transactions. If several transactions concurrently access an object, then making a new-master object or returning to the old master may be inappropriate because it commits or backs up all updates to the object by all transactions.

It is desirable to be able to commit or undo updates on a per-transaction basis. Given an action consistent state and a collection of in-progress transactions (i.e., commit not yet executed) one wants to be able to selectively undo a subset of the transactions without affecting the others. Such a facility is called in-progress transaction backup.

A second shortcoming of versions is that in the event of a media error, one must reconstruct the most recent consistent state. For example, if a page or collection of pages is lost from non-volatile storage, then they must be reconstructed from some redundant information. Doubly-recording the versions on independent devices is quite expensive for large objects. However, this is the technique used for some small objects such as the warm start file.

Lastly, writing a new version of a large database often consumes large amounts of storage and bandwidth.

Having abandoned the notion of versions, we adopt the approach of updating in place, and of keeping an incremental log of changes to the system state. (Logs are sometimes called audit trails or journals.)

Each action that modifies a recoverable object writes a log record giving the old and new value of the updated object. Read operations need generate no log records, but update operations must record enough information in the log so that, given the record at a later time, the operation can be completely undone or redone. These records will be aggregated by transaction, and collected in a common system log that resides in nonvolatile storage. The system log is duplexed and has independent failure modes.

In what follows we assume that the log never fails. By duplexing, triplexing, …., one can make this assumption less false.

Every recoverable operation must have:

- A DO entry that does the action and also records a log record sufficient to undo and to redo the operation,
- An UNDO entry that undoes the action given the log record written by the DO action,
- A REDO entry that redoes the action given the log record written by the DO action, and
- Optionally, a DISPLAY entry that translates the log into a human-readable format.

To give an example of an action and the log record it must write, consider the database record update operator. This action must record in the log the:

1. record name
2. the old record value (used for UNDO)
3. the new record value. (used for REDO)

The log subsystem augments this with the additional fields:

4. transaction identifier
5. action identifier
6. length of log record
7. pointer to previous log record of this transaction

```
DECLARE
     1 UPDATE_LOG_RECORD BASED,
          2 LENGTH      FIXED(16),   /*length of log record           */
          2 TYPE        FIXED(16),   /*code assigned to update log recs*/
          2 TRANSACTION FIXED(48),   /*name of transaction            */
          2 PREV_LOG_REC POINTER(31),/* relative address of prev log   */
                                     /*record of this transaction     */
          2 SET         FIXED(32),   /* name of updated set (table)    */
          2 RECORD      FIXED(32),   /*name of updated record         */
          2 NFIELDS     FIXED(16),   /*number of updated fields       */
          2 CHANGES (NFIELDS),       /*for each changed field:        */
               3 FIELD FIXED(16);    /* name of field                 */
               3 OLD_VALUE,          /*old value of field             */
                4 F_LENGTH FIXED(16),/*length of old field value      */
                4 F_ATOM CHAR (F_LENGTH), /*value in old field        */
               3 NEW_VALUE LIKE OLD_VALUE, /*new value of field       */
          2 LENGTH_AT_END FIXED(16); /*allows reading log backwards    */
```

The data manager's undo operation restores the record to its old value appropriately updating indices and sets. The redo operation restores the record to its new value. The display operation returns a text string giving a symbolic display of the log record.

The log itself is recorded on a dedicated media (disk, tape,...). Once a log record is recorded, it cannot be updated. However, the log component provides a facility to open read cursors on the log that will traverse the system log or will traverse the log of a particular transaction in either direction.

The UNDO operation must face a rather difficult problem at restart: The undo operation may be performed more than once if restart itself is redone several times (i.e. if the system fails during restart.) Also one may be called upon to undo operations that were never reflected in nonvolatile storage (i.e. log write occurred but object write did not.)

Similar problems exist for REDO. One may have to REDO an already done action if the updated object was recorded in non-volatile storage before the crash or if restart is restarted.

The write-ahead-log-protocol and high-water-marks solve these problems (see below).

5.8.3. 2. WRITE AHEAD LOG PROTOCOL

The recovery system postulates that memory comes in two flavors: volatile and nonvolatile storage. Volatile storage does not survive a system restart. Nonvolatile storage usually survives a system restart.

Suppose an object is recorded in non-volatile storage before the log records for the object are recorded in the non-volatile log. If the system crashes at such a point, then one cannot undo the update. Similarly, if the new object is one of a set that is committed together, and if a media error occurs on the object, then a mutually consistent version of the set of objects cannot be constructed from their non-volatile versions. Analysis of these two examples indicates that the log should be written to non-volatile storage before the object is written.

Actions are require to write log records whenever modifying recoverable objects. The leg (once recorded in nonvolatile storage) is considered to be very reliable. In general the log is dual recorded on physical media with independent failure modes (e.g., dual tapes or spindles) although single logging is a system option.

The Write Ahead Log Protocol (WAL) is:

- Before over-writing a recoverable object to nonvolatile storage with uncommitted updates, a transaction (process) should first force its undo log for relevant updates to nonvolatile log space.
- Before committing an update to a recoverable object, the transaction coordinator (see below) must force the redo and undo log to nonvolatile storage, so that it can go either way on the transaction commit. (This is guaranteed by recovery management that will synchronize the commit process with the writing of the phase-2 log transition record at the end of phase-1 of commit processing. This point cannot be understood before the section on two phase commit processing is read.)

This protocol needs to be interpreted broadly in the case of messages: One should not send a recoverable message before it is logged (so that the message can be canceled or retransmitted.) In this case, the wires of the network are the "non-volatile storage”.

The write-ahead-log protocol is implemented as follows. Every log record has a unique sequence number. Every recoverable object has a "high water mark" which is the largest log sequence number that applies to it. Whenever an object is updated, its high water mark is set to the log sequence number of the new log record. The object cannot be written to non-volatile storage before the log has been written past the object's high water mark. Log manager provides a synchronous call to force out all log records up to a certain sequence number.

At system restart a transaction may be undone or redone. If an error occurs the restart may be repeated. This means that an operation may be undone or redone more than once. Also, since the log is "ahead of" non-volatile storage the first undo may apply to an already undone (not-yet-done) change. Similarly the first redo may redo an already done change. This requires that the redo and undo operators be repeatable (idempotent) in the sense that doing them once produces the same result as doing them several times. Undo or redo may be invoked repeatedly if restart is retried several times or if the failure occurs during phase 2 of commit processing.

Here again, the high water mark is handy. If the high water mark is recorded with the object, and if the movement of the object to nonvolatile storage is atomic (this is true for pages and for messages) then one can read to high water mark to see if undo or redo is necessary. This is a simple way to make the undo and redo operators idempotent.

Message sequence numbers on a session perform the function of high water marks. That is the recipient can discard messages below the last sequence number received.

As a historical note, the need for WAL only became apparent with the widespread use of LSI memories. Prior to that time the log buffers resided in core storage that survived software errors, hardware errors and power failure. This allowed the system to treat the log buffers in core as non-volatile storage, At power shutdown, an exception handler in the data management dumps the log buffers. If this fails a scavenger is run which reads them out of core to storage. In general the contents of LSI storage does not survive power failures. To guard against power failure, memory failure and wild stores by the software, most systems have opted for the WAL protocol.

5.8.3.3. TWO PHASE COMMIT PROTOCOL

5.8.3.3.1. THE GENERALS PARADOX

In order to understand that the two-phase commit protocol solves some problem it is useful to analyze this generals paradox.

There are two generals on campaign. They have an objective (a hill) that they want to capture. If they simultaneously march on the objective they are assured of success. If only one marches, he will be annihilated.

The generals are encamped only a short distance apart, but due to technical difficulties, they can communicate only via runners. These messengers have a flaw, every time they venture out of camp they stand some chance of getting lost (they are not very smart.)

The problem is to find some protocol that allows the generals to march together even though some messengers get lost.

There is a simple proof that: no fixed length protocol exists: Let P be the shortest such protocol. Suppose the last messenger in P gets lost. Then either this messenger is useless or one of the generals doesn't get a needed message. By the minimality of P, the last message is not useless so one of the general doesn't march if the last message is lost. This contradiction proves that no such protocol P exists.

The generals paradox (which as you now see is not a paradox) has strong analogies to problems faced by a data recovery management when doing commit processing. Imagine that one of the generals is a computer in Tokyo and that the other general is a cash-dispensing terminal in Fuessen Germany. The goal is to

- open a cash drawer with a million marks in it (at Fuessen) and
- debit the appropriate account in the non-volatile storage of the Tokyo computer.

If only one thing happens, either the Germans or the Japanese will destroy the general that did not "march".

5.8.3.3.2. THE TWO PHASE COMMIT PROTOCOL

As explained above, there is no solution to the two generals problem.

If however, the restriction that the protocol have some finite fixed maximum length is relaxed then a solution is possible. The protocol about to be described may require arbitrarily many messages. Usually it requires only a few messages, sometimes it requires more and in some cases (a set of measure zero) it requires an infinite number of messages. The protocol works by introducing a commit coordinator. The commit coordinator has a communication path to all participants. Participants are either cohorts (processes) at several nodes or are autonomous components within a process (like DB and DC)or are both.

The commit coordinator asks all the participants to go into a state such that, no matter what happens, the participant can either redo or undo the transaction (this means writing the log in a very safe place).

Once the coordinator gets the votes from everyone:

- If anyone aborted, the coordinator broadcasts abort to all participants, records abort in his log and terminates. In this case all participants will abort.
- If all participants voted yes, the coordinator synchronously records a commit record in the log, then broadcasts commit to all participants and when an acknowledge, is received from each participant, the coordinator terminates.

The key to the success of this approach is that the decision to commit has been centralized in a single place and is not time constrained.

The following diagrams show the possible interactions between a coordinator and a participant. Note that a coordinator may abort a participant that agrees to commit. This may happen because another participant has aborted

```
      COORDINATOR                          PARTICIPANT
commit
  ------>             request commit
                 ------------------------>

                        agree
                <-----------------------

                        commit
                ----------------------->
    yes                                             commit
 <-------                                      --------->
```

(1) Successful commit exchange.

```
commit
  ------>             request commit
                 ------------------------>

                        abort
                <-----------------------

    no                                              abort
 <-------                                      --------->
```

(2) Participant aborts commit.

```
commit
  ------>             request commit
                 ------------------------>

                        agree
                <-----------------------

                        abort
                ----------------------->
    no                                              abort
 <-------                                      --------->
```

(3)Coordinator aborts commit.

Three possible two-phase commit scenarios.

The logic for the coordinator is best described by a simple program:

```
COORDINATOR: PROCEDURE;
    VOTE='COMMIT'; /*collect votes */
    DO FOR EACH PARTICIPANT WHILE(VOTE='COMMIT');
        DO;
        SEND HIM REQUEST_COMMIT;
        IF REPLY != 'AGREE' THEN VOTE='ABORT';
        END;
    IF VOTE='COMMIT' THEN
        DO; /*if all agree then commit+/
        WRITE_LOG(PHASE12_COMMIT)FORCE;
        FOR EACH PARTICIPANT;
            DO UNTIL (+ACK);
            SEND HIM COMMIT;
            WAIT +ACKNOWLEDGE;
            IF TIME LIMIT THEN RETRANSMIT;
            END;
        END;
        ELSE
            DO; /*if any abort, then abort*/
            FOR EACH PARTICIPANT
                DO UNTIL (+ACK);
                SEND MESSAGE ABORT;
                WAIT +ACKNOWLEDGE;
                IF TIME_LIMIT THEN RETRANSMIT;
                END
            END;
        WRITE_LOG(COORDINATOR_COMPLETE);/*common exit*/
        RETURN;
        END COORDINATOR;
```

The protocol for the participant is simpler:

```
PARTICIPANT: PROCEDURE;
        WAIT FOR REQUEST_COMMIT; /*phase 1 */
        FORCE UNDO REDO LOG TO NONVOLATILE STORE;
        IF SUCCESS THEN /*writes AGREE in log */
                REPLY 'AGREE';
        ELSE
                REPLY 'ABORT';
        WAIT FOR VERDICT; /*phase2 */
        IF VERDICT ='COMMIT' THEN
                DO;
                RELEASE RESOURCES & LOCKS;
                REPLY +ACKNOWLEDGE;
                END;
        ELSE
                DO;
                UNDO PARTICIPANT;
                REPLY +ACKNOWLEDGE;
                END;
        END PARTICIPANT;
```

There is a last Piece of logic that needs to be included: In the event of restart, recovery manager has only the log and the nonvolatile store. If the coordinator crashed before the PHASE12_COMMIT record appeared in the log, then restart will broadcast abort to all participants. If the transaction's PHASE12_COMMIT record appeared and the COORDINATOR_COMPLETE record did not appear, then restart will re-broadcast the COMMIT message. If the transaction's COORDINATOR_COMPLETE record appears in the log, then restart will ignore the transaction. Similarly transactions will be aborted if the log has not been forced with AGREE. If the AGREE record appears, then restart asks the coordinator whether the transaction committed or aborted and acts accordingly (redo or undo.)

Examination of this protocol shows that transaction commit has two phases:

1. Before its PHASE12_COMMIT or AGREE_COMMIT log record has been written and,
2. After its PHASE12_COMMIT or AGREE_COMMIT log record has been written.

This is the reason it is called a two-phase commit protocol. A fairly lengthy analysis is required to convince oneself that a crash or lost message will not cause one participant to “march" the wrong way.

Let us consider a few cases. If any participant aborts or crashes in his phase 1 then the entire transaction will be aborted (because the coordinator will sense that he is not replying using timeout).

If an participant crashes in his phase 2 then recovery manager, as a part of restart of that participant, will ask the coordinator whether or not to redo or undo the transaction instance. Since the participant wrote enough information for this in the log during phase 1, recovery manager can go either way on completing this participant. This requires that the undo and redo be idempotent operations. Conversely, if the coordinator crashes before it writes the log record, then restart will broadcast abort to all participants. No participant has committed because the coordinator’s PHASE12_COMMIT record is synchronously written before any commit messages are sent to participants. On the other hand if the coordinator's PHASE12_COMMIT record is found in the log at restart, then the recovery manager broadcasts commit to all participants and waits for acknowledge. This redoes the transaction (coordinator).

This rather sloppy argument can be (has been) made more precise. The net effect of the algorithm is that either all the participants commit or that none of them commit (all abort.)

5.8.3.3.3. NESTED TWO PHASE COMMIT PROTOCOL

Many optimizations of the two-phase commit protocol are possible. As described above, commit requires 4N messages if there are N participants. The coordinator invokes each participant once to take the vote and once to broadcast the result. If invocation and return are expensive (e.g., go over thin wires) then a more economical protocol may be desired.

If the participants can be linearly ordered then a simpler and faster commit protocol that has 2N calls and returns is possible. This protocol is called the nested two-phase commit. The protocol works as follows:

- Each participant is given a sequence number in the commit call order.
- In particular, each participant knows the name of the next participant and the last participant knows that he is the last.

Commit consists of participants successively calling one another (N-l calls) after performing phase 1 commit. At the end of the calling sequence each participant will have successfully completed phase 1 or some participant will have broken the call chain. So the last participant can perform phase 2 and returns success. Each participant keeps this up so that in the end there are N-l returns to give a grand total of 2(N-1) calls and returns on a successful commit. There is one last call required to signal the coordinator (last participant)that the commit completed so that restart can ignore redoing this transaction. If some participant does not succeed in phase 1 then he issues abort and transaction undo is started.

The following is the algorithm of each participant:

```
COMMIT: PROCEDURE;
    PERFORM PHASE 1 COMMIT;
    IF FAIL THEN RETURN FAILURE;
    IF I_AM_LAST THEN
        WRITE_LOG(PHASE12)FORCE;
        ELSE
            DO;
            CALL COMMIT(I+l);
            IF FAIL THEN
                DO;
                ABORT;
                RETURN FAILURE;
                END;
            END;
        PERFORM PHASE 2 COMMIT;
        IF I_AM_FIRST THEN
                INFORM LAST THAT COMMIT COMPLETED;
        RETURN SUCCESS;
        END;
```

The following gives a picture of a three deep nest:

```
            Rl
commit.
--------->
               --PHASE1 --> R2
                              --PHASEl -> R3
                             <--PHASE2 --
               <--PHASE2 --
  yes                     Fin
<---------      ------------------------->
```

(a) a successful commit.

```
            Rl
commit
--------->
               --Phase1 --> R2
                              --PHASEl--> R3
                              <--ABORT --
               <--ABORT --
      no
<--------
```

(b) an unsuccessful commit.

5.8.4.3.3.. COMPARISON BETWEEN GENERAL AND NESTED PROTOCOLS

The nested protocol is appropriate for a system in which

- . The message send-receive cost is high and broadcast not available.
- . The need for concurrency within phase 1 and concurrency within phase 2 is low,
- . The participant and cohort structure of the transaction is static or universally known.

Most data management systems have opted for the nested commit protocol for these reasons. On the other hand the general two phase commit protocol is appropriate if:

- Broadcast is the normal mode of interprocess communication (in that case the coordinator sends two messages and each process sends two messages for a total of 2N messages.)Aloha net, Ethernet, ring-nets, and space-satellite nets have this property.
- Parallelism among the cohorts of a transaction is desirable (the nested protocol has only one process active at a time during commit processing.)

5.8.3.4. SUMMARY OF RECOVERY PROTOCOLS

- The consistency lock protocol isolates transactions from inconsistencies due to concurrency.
- The DO-REDO-UNDO log record protocol allows for and uncommitted actions.
- The write-ahead-log protocol insures that the log is ahead of nonvolatile storage, so that undo and redo can always be performed.
- The two-phase commit protocol coordinates the commitment of autonomous participants (or cohorts) within a transaction.

The following table explains the virtues of the write-ahead-log and two-phase-commit protocols. It examines the possible situations after a crash. The relevant issues are whether an update to the object survived (was written to nonvolatile storage), and whether the log record corresponding to the update survived. One mill never have to redo an update whose log record is not written because: Only committed transactions are redone, and COMMIT writes out the transaction's log records before the commit completes. So the (no, no, redo) case is precluded by two-phase commit. Similarly, write-ahead-log (WAL) precludes the (no, yes,*) cases, because an update is never written before its log record. The other cases should be obvious.

LOG RECORD WRITTEN	OBJECT WRITTEN	REDO OPERATION	UNDO OPERATION
NO	NO	IMPOSSIBLE BECAUSE OF TWO PHASE COMMIT	NONE
NO	YES	IMPOSSIBLE BECAUSE OF WRITE AHEAD LOG	
YES	NO	REDO	NONE
YES	YES	NONE	UNDO

5.8.4. STRUCTURE OF RECOVERY MANAGER

Recovery management consists of two components.

- Recovery manager that is responsible for tracking transactions and the coordination of transaction COMMIT and ABORT, and system CHECKPOINT and RESTART (see below).

- Log manager that is used by recovery manager and other components to record information in the system log for the transaction or system.

```
+----------+                    +----------+
| RECOVERY |      UNDO        +--|--------+ |
| MANAGER  | ----------->     |  OTHER    | |
+----------+      REDO        +  ACTIONS  |-+
    |                         +-----------+
    |                              |
    |         READ & WRITE LOG     |
    +--------------+---------------+
                   |
           +-------------+
           |     LOG     |
           |   MANAGER   |
           +-------------+
```

Relationship between Log manager and component actions.

The purpose of the recovery system is two-fold: First, the recovery system allows an in-progress transaction to be “undone” in the event of a “minor” error, without affecting other transactions. Examples of such errors are operator cancellation of the transaction, deadlock, timeout, protection or integrity violation, resource limit,..,.

Second, in the event of a “serious” error, the recovery subsystem minimizes the amount of work that is lost and by restoring all data to its most recent committed state. It does this by periodically recording copies of key portions of the system state in nonvolatile storage, and by continuously maintaining a log of changes to the state, as they occur. In the event of a catastrophe, the most recent transaction consistent version of the state is reconstructed from the current state on nonvolatile storage by using the log to

- undo any transactions that were incomplete at the time of the crash.
- redo any transactions that completed in the interval between the checkpoints and the crash.

In the case that on-line nonvolatile storage does not survive, one must start with an archival version of the state and reconstruct the most recent consistent state from it. This process requires:

- Periodically making complete archive copies of objects within the system.
- Running a change accumulation utility against the logs written since the dump. This utility produces a much smaller list of updates that will bring the image dump up to date. Also this list is sorted by physical address so that adding it to the image dump is a sequential operation.
- The change accumulation is merged with the image to reconstruct the most recent consistent state.

  Other reasons for keeping a lag of the actions of transactions include auditing and performance monitoring since the log is a trace of system activity.

  There are three separate recovery mechanisms:

  1. Incremental log of updates to the state.
  2. Current on-line version of the state.
  3. Archive versions of the state.

5.8.4.1. TRANSACTION SAVE LOGIC

When the transaction invokes SAVE, a log record is recorded which describes the current state of the transaction. Each component involved in the transaction is then invoked, and it must record whatever it needs to restore its recoverable objects to their state at this point. For example, the terminal handler might record the current state of the session so that if the transaction backs up to this point, the terminal can be reset to this point. Similarly, database manager might record the positions of cursors. The application program may also record log records at a save point.

A save point does not commit any resources or release any locks.

5.8.4.2. TRANSACTION COMMIT LOGIC

When the transaction issues COMMIT, recovery manager invokes each component (participant) to perform commit processing. The details of commit processing were discussed under the topics of recovery protocols above. Briefly, commit is a two-phase process. During phase 1, each manager writes a log record that allows it to go either way on the transaction (undo or redo). If all resource managers agree to commit, then recovery manager forces the log to secondary storage and enters phase 2 of commit. Phase 2 consists of committing updates: sending messages, writing updates to nonvolatile storage and releasing locks.

In phase 1 any resource manager can unilaterally abort the transaction thereby causing the commit to fail. Once a resource manager agrees to phase 1 commit, that resource manager must be willing to accept either abort or commit from recovery manager.

5.8.4.3. TRANSACTION BACKUP LOGIC

The effect of any incomplete transaction can be undone by reading the log of that transaction backwards undoing each action in turn. Given the log of a transaction T:

```
UNDO(T):
     DO WHILE (LOG(T)!= NULL);
          LOG_RECORD = LAST_ RECORD (LOG(T));
          UNDOER = WHO_WROTE(LOG_ RECORD);
          CALL UNDOER(LOG_RECORD);
          INVALIDATE(LOG_RECORD);
          END UNDO;
```

Clearly, this process can be stopped half-day, thereby returning the transaction to an intermediate save point. Transaction save points allow the transaction to backtrack in case of some error and yet salvage all successful work.

From this discussion it follows that a transaction's log is a push down stack, and that writing a new record pushes it onto the stack, while undoing a record pops it off the stack (invalidates it). For efficiency reasons, all transaction logs are merged into one system log that is then mapped into a log file. But, the log records of a particular transaction are threaded together and anchored off of the process executing the transaction.

Notice that UNDO requires that while the transaction is active, the log must be directly addressable. This is the reason that at least one version of the log should be on some direct access device. A tape-based log would not be convenient for in-progress transaction undo for this reason (tapes are not randomly accessed).

The undo logic of recovery manager is very simple. It reads a record, looks at the name of the operation that wrote the record and calls the undo entry point of that operation using the record type. Thus recovery manager is table driven and therefore it is fairly easy to add new operations to the system.

Another alternative is to defer updates until phase 2 of commit processing. Once a transaction gets to phase 2, it must complete successfully, thus if all updates are done in phase 2 no undo is ever required (redo logic is required.) IMS data communications and IMS Fast Path use this protocol.

5.8.4.4. SYSTEM CHECKPOINT LOGIC

System checkpoints may be triggered by operator commands, timers, or counters such as the number of bytes of log record since last checkpoint. The general idea is to minimize the distance one must travel in the log in the event of a catastrophe. This must be balanced against the cost of taking frequent checkpoints. Five minutes is a typical checkpoint interval.

Checkpoint algorithms that require a system quiesce should be avoided because they imply that checkpoints will be taken infrequently thereby making restart expensive.

The checkpoint process consists of writing a BEGIN_CHECKPOINT record in the log, then invoking each component of the system so that it can contribute to the checkpoint, and then writing an END_CHECKPOINT record in the log. These records bracket the checkpoint records of the other system components. Such a component may write one or more log records so that it will be able to restart from the checkpoint. For example, buffer manager will record the names of the buffers in the buffer pool, file manager might record the status of files, network manager may record the network status, and transaction manager will record the names of all transactions active at the checkpoint.

After the checkpoint log records have been written to non-volatile storage, recovery manager records the address of the most recent checkpoint in a warm start file. This allows restart to quickly locate the checkpoint record (rather than sequentially searching the log for it.) Because this is such a critical resource, the restart file is duplexed (two copies are kept) and writes to it are alternated so that one file points to the current and another points to the previous checkpoint log record.

At system restart, the programs are loaded and the transaction manager invokes each component to re-initialize itself. Data communications begins network-restart and the database manager reacquires the database from the operating system (opens the files).

Recovery manager is then given control. Recovery manager examines the most recent warm start file written by checkpoint to discover the location of the most recent system checkpoint in the log. Recovery manager then examines the most recent checkpoint record in the log. If there was no work in progress at the system checkpoint and the system checkpoint is the last record in the log then the system is in restarting from a shutdown in a quiesced state. This is a warm start and no transactions need be undone or redone. In this case, recovery manager writes a restart record in the log and returns to the scheduler, which opens the system for general use.

On the other hand if there was work in progress at the system checkpoint, or if there are further leg records then this is a restart from a crash (emergency restart).

The following figure will help to explain emergency restart logic:

```
T1  |---------|          +                                  <
T2     |-----------------+-------|                          <
T3                       +          |----------------|      <
T4  |--------------------+----------------------------------<
T5                       +                          |------<
                   CHECKPOINT                            SYSTEM
                                                          CRASH
```

Five transaction types with respect to the most recent system checkpoint and the crash point. Transactions T1, T2, and T3 have committed and must be redone. Transactions T4 and T5 have not committed and so must be undone. Let's call transactions like T1, T2 and T3 winners and lets call transactions like T4 and T5 losers. Then the restart logic is:

```
RESTART: PROCEDURE;
     DICHOTOMIZE WINNERS AND LOSERS;
     REDO THE WINNERS;
     UNDO THE LOSERS;
     END RESTART;
```

It is important that the REDOs occur before the UNDO (Do you see why (we are assuming page-locking and high-water marks from log-sequence numbers?)

As it stands, this implies reading every log record ever written because redoing the winners requires going back to redo almost all transactions ever run.

Much of the sophistication of the restart process is dedicated to minimizing the amount of work that must be done, so that restart can be as quick as possible, (We are describing here one of the more trivial workable schemes.) In general restart discovers a time T such that redo log records written prior to time T are not relevant to restart.

To see how to compute the time T, we first consider a particular object: a database page P. Because this is a restart from a crash, the most recent version of P may or may not have been recorded on non-volatile storage. Suppose page P was written out with high water mark LSN(P). If the page was updated by a winner "after" LSN(P), then that update to P must be redone. Conversely, if P was written out to nonvolatile storage with a loser's update, then those updates must be undone. (Similarly, message M may or may rot have been sent to its destination.) If it was generated by a loser, then the message should be canceled. If it was generated by a committed transaction but not sent then it should be retransmitted.) The figure below illustrates the five possible types of transactions at this point: T1 began and committed before LSN(P), T2 began before LSN(P) and ended before the crash, T3 began after LSN(P)and ended before the crash, T4 began before LSN(P) but its COMMIT record does not appear in the log, and T5 began after LSN(P) and apparently never ended. To honor the commit of T1, T2 and T3 requires that their updates be added to page P (redone). But T4, T5, and T6 have not committed and so must be undone.

```
T1  |---------|          +                                  <
T2     |-----------------+-------|                          <
T3                       +         |----------------|       <
T4  |--------------------+----------------------------------<
T5                       +                           |------<
                   wrote Page P                            SYSTEM
                    with LSN(P)                            CRASH
```

Five transactions types with respect to the most recent write of page P and the crash point,

Notice that none of the updates of T5 are reflected in this state so T5 is already undone. Notice also that all of the updates of T1 are in the state so it need not be redone. So only T2, T3, and T4 remain. T2 and T3 must be redone from LSN(P) forward. The updates of the first half of T2 are already reflected in the page P because it has log sequence number LSN(P). On the other hand, T4 must be undone from LSN(P) backwards. (Here we are skipping over the following anomaly: if after LSN(P), T2 backs up to a point prior to the LSN(P)then some undo work is required for T2. This problem is not difficult, just annoying.)

Therefore the oldest <u>redo</u> log record relevant to P is at or after LSN(P). (The write-ahead-log protocol is relevant here.) At system checkpoint, data manager records <u>MINLSN</u>, the log sequence number of the oldest page not yet written (the minimum LSN(P)of all pages, P, not yet written.) Similarly, transaction manager records the name of each transaction active at the checkpoint. Restart chooses T as the MINLSN of the most recent checkpoint.

Restart proceeds as follows: It reads the system checkpoint log record and puts each transaction active at the checkpoint into the loser set.

It then scans the log forward to the end. If a COMMIT log record is encountered, that transaction is promoted to the winners set. If a BEGIN_TRANSACTION record is found, the transaction is tentatively added to the loser set. When the end of the log is encountered, the winners and losers have been computed. The next thing is to read the log forwards from MINLSN, redoing the winners. Then it starts from the end of the log, read the log backwards undoing the losers.

This discussion of restart is very simplistic. Many systems have added mechanisms to speed restart by:

- Never write uncommitted objects to non-volatile storage (stealing) so that undo is never required.
- Write committed objects to secondary storage at phase 2 of commit (forcing), so that redo is only rarely required (this maximizes “MINLSN).
- Log the successful completion of a write to secondary storage. This minimizes redo.
- Force all objects at system checkpoint, thereby maximizing MINLSN.

<u>5.8.4.5. MEDIA FAILURE LOGIC</u>

In the event of a hard system error (one non-volatile storage integrity), there must be minimum of lost work. Redundant copies of the object must be maintained, for example on magnetic tape that is stored in a vault. It is important that the archive mechanism have independent failure modes from the regular storage subsystem. Thus, using doubly redundant disk storage would protect against a disk head crash, but wouldn't protect against a bug in the disk driver routine or a fire in the machine room.

The archive mechanism periodically writes a checkpoint of the data base contents to magnetic tape, and writes a redo log of all update actions to magnetic tape, Then recovering from a hard failure is accomplished by locating the most recent surviving version on tape, loading it back into the system, and then redoing all updates from that point forward using the surviving log tapes.

While performing a system checkpoint causes relatively few disk writes, and takes only a few seconds, copying the entire database to tape is potentially a lengthy operation. Fortunately there is a (little used) trick: one can take a fuzzy dump or an object by writing it to archive with an idle task. After the dump is taken, the log generated during the fuzzy dump is merged with the fuzzy dump to produce a sharp dump. The details of this algorithm are left as an exercise for the reader.

5.8.4.6. COLD START LOGIC

Cold start is too horrible to contemplate. Since we assumed that the log never fails, cold start is never required. The system should be cold started once: when the implementers create its first version. Thereafter, it should be restarted. In particular moving to new hardware or adding to a new release of the system should not require a cold start. (i.e. all data should survive.) Note that this requires that the format of the log never change, it can only be extended by adding new types of log records.

5.8.5. LOG MANAGEMENT

The log is a large linear byte space. It is very convenient if the log is write-once, and then read-only. Space in the log is never re-written. This allows one to identify log records by the relative byte address of the last byte of the record.

A typical (small) transaction writes 500 bytes of log. One can run about one hundred such transactions per second on current hardware. There are almost 100,000 seconds in a day. So the log can grow at 5 billion bytes per day. (more typically, systems write four log tapes a day at 50 megabytes per tape.) Given those statistics the log addresses should be about 48 bits long (good for 200 years on current hardware.)

Log manager must map this semi-infinite logical file (log) into the rather finite files (32 bit addresses) provided by the basic operating system. As one file is filled, another is allocated and the old one is archived. Log manager provides other resource managers with the operations:

WRITE_LOG: causes the identified log record to be written to the log. Once a log record is written. It can only be read. It cannot be edited. WRITE_LOG is the basic command used by all resource managers to generate log records. It returns the address of the last byte of the written log record.

FORCE-LOG: causes the identified log record and all prior log records to be recorded in nonvolatile storage. When it returns, the writes have completed.

OPEN-LOG: indicates that the issuer wishes to read the log of some transaction, or read the entire log in sequential order. It creates a read cursor on the log.

SEARCH-LOG: moves the cursor a designated number of bytes or until a log record satisfying some criterion is located.

READ-LOG: requests that the log record currently selected by the log cursor be read.

CHECK-LOG: allows the issuer to test whether a record has been placed in the non-volatile log and optionally to wait until the log record has been written out.

GET-CURSOR: causes the current value of the write cursor to be returned to the issuer. The RBA (relative byte address) returned may be used at a later time to position a read cursor.

CLOSE-LOG: indicates the issuer is finished reading the log.

The write log operation moves a new log record to the end of the current log buffer. If the buffer fills, another is allocated and the write continues into the new buffer.

When a log buffer fills or when a synchronous log write is issued, a log daemon writes the buffer to nonvolatile storage. Traditionally, logs have been recorded on magnetic tape because it is so inexpensive to store and because the transfer rate is quite high. In the future disk, CCD (nonvolatile?) or magnetic bubbles may be attractive as a staging device for the log. This is especially true because an on-line version of the log is very desirable for transaction undo and for fast restart.

It is important to doubly record the log. If the log is not doubly recorded, then a media error on the log device will produce a cold start of the system. The dual log devices should be on separate paths so that if one device or path fails the system can continue in degraded mode (this is only appropriate for applications requiring high availability.)

The following problem is left as an exercise for the reader: We have decided to log to dedicated dual disk drives. When a drive fills it will be archived to a mass storage device. This archive process makes the disk unavailable to the log manager (because of arm contention.) Describe a scheme which:

- minimizes the number of drives required, .
- always has a large disk reserve of free disk space, and
- always has a large fraction of the recent section of the log on line.

5.8.5.1. LOG ARCHIVE AND CHANGE ACCUMULATION

When the log is archived, it can be compressed so that it is convenient for media recovery. For disk objects, log records can be sorted by cylinder, then track then sector then time. Probably, all the records in the archived log belong to completed transactions. So one only needs to keep redo records of committed (not aborted) transactions. Further only the most recent redo record (new value) need be recorded. This compressed redo log is called a change accumulation log. Since it is sorted by physical address, media recover becomes a merge of the image dump of the object and its change accumulation tape.

```
FAST_MEDIA_RECOVERY: PROCEDURE(IMAGE, CHANGE_ACCUMULATION_LOG);
     DO WHILE ( ! END_OF_FILE IMAGE);
          READ IMAGE PAGE;
          UPDATE WITH REDO RECORDS FROM CHANGE_ACCUMULATION_LOG;
          WRITE IMAGE PAGE TO DISK;
          END
     END;
```

This is a purely sequential process (sequential on input files and sequential on disk being recovered) and so is limited only by the transfer rates of the devices.

The construction of the change accumulation file can be done off-line as an idle task.

If media errors are rare and availability of the data is not a critical problem then one may run the change accumulation utilities when needed. This may save building change accumulation files that are never used.

5.8.6. EXAMPLES OF RECOVERY ROUTINES.

5.8.6.1. HOW TO GET PERFECTLY RELIABLE DATA COMMUNICATIONS

Watch this space (a coming attraction)

5.8.6.1. HOW TO GET PERFECTLY RELIABLE DATA MANAGEMENT

Watch this space (a coming attraction)

5.8.7. HISTORICAL NOTE ON RECOVERY MANAGEMENT.

Most of my understanding of the topic of recovery derives from the experience of the IMS developers and from the development of System R. Unfortunately, both these groups have been more interested in writing code and understanding the problems, than in writing papers. Hence, there is very little public literature that I can cite. Ron Obermark seems to have discovered the notion of write-ahead-log in 1974. He also implemented the nested-two-phase commit protocol (almost). This work is known as the IMS-Program Isolation Feature. Earl Jenner and Steve Weick first documented the two-phase commit protocol in 1975, although it seems to have its roots in some systems built by Niko Garzado in 1970. Subsequently, the SNA architects, the IMS group, and the System R group has explored various implementations of these ideas. Paul McJones (now at Xerox) and I were stimulated by Warren Titlemann's history file in INTERLISP to implement the DO-UNDO-REDO paradigm for System R. The above presentation of recovery derives from drafts of various (unpublished) papers co-authored with John Nauman, Paul McJones, and Homer Leonard. The two-phase-commit protocol was independently discovered by Lampson and Sturgis (see below) and the nested commit protocol was independently discovered by Lewis, Sterns, and Rosenkrantz (see below.)

5.8.8 BIBLIOGRAPHY

Alsberg, “A Principle for Resilient Sharing of Distributed Resources,” Second National Conference on Software Engineering, IEEE Cat. No. 76CH1125-4C, 1976, pp. 562-570. (A novel proposal (not covered in these notes) which describes a protocol whereby multiple hosts can cooperate to provide a reliable transaction processing. It is the first believable proposal for system duplexing or triplexing I have yet seen. Merits further study and development.)

Bjork, "Recovery Scenario for a DB/DC System, I1 Proceedings ACM National Conference, 1973, pp. 142-146.

Davies, “Recovery Semantics for a DB/DC System," Proceedings ACM National Conference, 1973, pp. 136-141. (The above two companion papers are the seminal work in the field. Anyone interested in the topic of software recovery should read them both at least three times and then once a year thereafter.)

Lampson, Sturgis, “Crash Recovery in a Distributed System," Xerox Palo Alto Research Center, 1976 To appear in CACM. (A very nice paper which suggests the model of errors presented in section 5.8.1 and goes on to propose a three-phase commit protocol. This three-phase commit protocol is an elaboration of the two-phase commit protocol. This is the first (only) public mention of the two-phase commit protocol.)

Rosenkrantz, Sterns, Lewis, "System Level Concurrency Control for Data Base Systems, General Electric Research, Proceedings of Second Berkeley Workshop on Distributed Data Management and Data Management, Lawrence Berkeley Laboratory, LBL-6146, 1977, pp. 132-145. also, to appear in Transactions on Data Systems, AC& (Presents a form of nested commit protocol, allows only one cohort at a time to execute.)

Information Management System/Virtual Storage (IMS/VS), System Programming Reference Manual, IBM Form No SH20-9027-2, p. 5-2. (Briefly describes WAL.)