AUTHORIZATION AND TRANSACTION MANAGEMENT IN DISTRIBUTED DATABASE SYSTEMS

George Pluimakers

RUU-CS-88-8
March 1988

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands

AUTHORIZATION AND TRANSACTION MANAGEMENT IN DISTRIBUTED DATABASE SYSTEMS

George Pluimakers

Technical Report RUU-CS-88-8
March 1988

Department of Computer Science
University of Utrecht
P.O.Box 80.012, 3508 TA Utrecht
The Netherlands

What's that Machine noise
It's bytes and megachips for tea
It's that Machine, boys
With Random Access Memory
Never worry, never mind
Not for money, not for gold.

It's software it's hardware
It's heartbeat is time-share
It's midwife's a disk drive
It's sex-life is quantised
It's self-perpetuating a parahumanoidarianised.

from MACHINES (or BACK TO HUMANS),

song of popgroup QUEEN.

Preface

With this Master's Thesis I close my study in Mathematics and Computer Science at the University of Utrecht. I thank my advisor Jan van Leeuwen for suggesting the subject of this paper and pointing out many improvements of the manuscript.
Further I would like to thank my friend Paul Gorissen for providing a macro-set for nroff, a text formatter for the UNIX system, with the aid of which the lay-out is greatly improved.

Utrecht, october 1985.

Abstract

This master's thesis covers two topics on distributed database sytems, in which security is involved:

- authorization :the dynamical granting, checking and revocation of access privileges to permit users to share data on the one hand, and to retain the possibility to restrict such privileges on the other hand.
- transaction management: the control of the execution of the database actions comprising a transaction in such a way that the transaction has the properties mentioned above.

For both subjects I give a survey of the various problems arising in the area. Then algorithms to solve these problems will be described. I aim at being as concrete as possible, when explaining an algorithm. Therefore the solving algorithms are given with the aid of protocols, procedure bodies, storage actions or other concrete actions to be taken.
Distributed problems are very often more difficult than their non-distributed equivalents. Much attention will be paid to how the algorithms deal with the distributed aspect.

Address of Author: NCR Systems Engineering, P.O. Box 492, 3430 AL Nieuwgein, the Netherlands

CONTENTS

Introduction

Basically a database system is nothing more than a computer-based record keeping system whose overall purpose is to record and maintain information [Date 81]. A database system has four components: data, hardware, software and users.
The data stored in the system is partitioned into one or more databases. In general a database is integrated (a unification of several distinct datafiles) and shared (pieces of data may be accessed by several users). We will also allow concurrent sharing (some users access the same piece of data, probably at the same time).
The hardware consists of the secondary storage media: disks, drums etc. on which the database resides together with the associated devices (control units, channels, etc.).
Definition 1
A database management system (DBMS) is a layer of software between the physical database and the users.
End of Definition

All requests from users for access are handled by the DBMS, hiding the hardware from the users.
There are five classes of user:
- database administrator (DBA): defines and installs the system and makes policy decisions about the operation of the system
- database operator: handles the operation of the system, manages system startup, shutdown and responds to user requests
- database programmer: installs and maintains the underlying operating system of the DataBase-DataCommunication system (DB-DC)
- application programmer: defines and implements new application programs to be used by end users and the DBA
- end user: uses the system to enter or retrieve data.

The basic units about which data is recorded in the database are called entities. Each entity is represented within the system by one or more objects which are identified by (name, node) pairs, where 'name' is the entity name and 'node' is the place at which the object is located. At each instant an object has a value.
In general there are relationships or associations among the entities.
Definition 2
An action is a call to the DBMS which is implemented in such a way that it will be atomic : it either completes or it does not complete and has no effect at all on the database.
End of Definition

Thus, if two processes concurrently perform actions the effect will be as though one of the actions were performed before the other. The system translates a request against an entity into one or more actions on objects.
Definition 3
A transaction is a delimited sequence of database actions which together form a logical unit of work, which is also a unit of
1) recovery: after recovery from a crash either all actions of the

transaction will have been executed or the transaction has no effect at all. This is called the atomicity property for transactions.
2) locking: the locks on objects to be accessed in the transaction are all released at the end of a transaction.
3) consistency: a transaction transforms a consistent system state (to be defined later) into a new consistent state.

End of Definition

Definition 4
An application program defines the sequences of actions which constitute a transaction. It contains statements in a certain programming language plus database requests.
End of Definition

This paper is about distributed databases.
Definition 5
A distributed database is not stored as a whole at a single physical location, but rather is spread across a network of autonomous computers that are geographically dispersed and connected via communication links. The computers in the network are called nodes, their locations are called sites. Often these notions are treated as synonyms.
End of Definition

Definition 6
A distributed database system (DDBS) is a layer of software between a physical distributed database and the users.
End of Definition

Distributed data accesses require cooperation among some independent nodes. Via the network a node is in conversation with the nodes at other sites. Setting up such a conversation between two nodes involves exchanging information, agreeing on protocols, formats, flow control, cryptographic techniques (to prevent from eavesdropping) and, last but not least, authentication to check that thay are indeed conversing with each other. Once a conversation is established it is assumed that the systems subsequently trust each other, and are communicating over a secure link. Message encryption should be used if the link is insecure.
The network is expected to be unreliable and relatively slow. Ideally the system should be able to:
- resist delayed messages
- resist nodes that went down
- permit inhomogeneity of the nodes (e.g. the nodes differ in data management, internal data structure representation or something else).

For distributed database sytems the notions of action, transaction, and application program are defined the same as before.

CHAPTER I

AUTHORIZATION

I.1. Preliminary remarks and definitions

A multi-user data base system, whether or not it is distributed, must permit users to selectively share data. On the other hand the ability to restrict data access must be retained. There has to be a mechanism to provide protection and secrecy, so as to provide access to properly authorized users only.

Definition I.1.1

Authentication of a user A is the verification of A's claimed identity.

End of Definition

Authentication is required whenever a user A wishes to gain access to a system over a direct, i.e., trusted, carrier. For more details see [PlvL 85a,b]. Authentication is closely related to authorization.

Definition I.1.2

Authorization of a user A by user B means that

1) B authenticates A
2) B verifies, grants and monitors certain rights that A claims (to have)

End of Definition

In the case of distributed data base systems the rights will be access rights, also called privileges. A user can issue two commands on an access right he owns:

- GRANT, by which he "grants" the right to another user. This command enables the user to share an access right.
- REVOKE, with which he withdraws the right from a user to which it was previously granted.

The scenario of this chapter is that a user, known by some unique USERID is logged on to his local DBMS, after being authenticated by (a combination of) his password, the location of terminal and the time of day. The authentication results in a role for the user. This means that the user is incorporated in one of the five classes of users (see page i).
Once a person establishes a role he is authorized to perform certain transactions on the system. The concept of role serves the purpose of grouping users of the system together, thereby decomposing authorization into the following two questions:

- what should the authorization of a role be?
- who should be authorized to use the role?

When a user issues a request to his local DBMS that involves a remote DBMS request, the local DBMS sets up a conversation with the remote DBMS, and passes along his USERID. The receiving remote DBMS will act as if that USERID were directly logged in to the system. Thus each site performs local authorization checking on its own local data. This greatly helps in achieving location transparency: though data is geographically distributed and may move, the programmer's viewpoint is as if all data is in one node. Each site keeps access control information about its local data, and checks authorization upon every access. Authorization is by USERID which therefore must be unique over

the network.

I.2. The local access control mechanism

In current single site data base management systems the ability to grant authorization to perform actions on objects resides with a central database administrator or with the creator of the object. Many systems use password schemes for this purpose. Such systems have several disadvantages :

1) passwords are vulnerable to guessing
2) the password is disclosed when a grant is made, thereby making authorization external to the DBMS; thus executing a REVOKE command becomes difficult
3) password schemes don't permit data-dependent access control

I describe the problems of dynamically authorizing data-independent and data-dependent operations, and of revoking such authorization, in an environment in which more than one user may grant privileges on the same object. I distinguish between relational and hierarchical databases.

I.3. The relational database case

In this case the basic objects are relations, which are sets of n-tuples.

Example I.3.1

EMPLOYEE(NAME, SALARY, MANAGER, DEPT).

This relation has a tuple (row) for each employee, giving his name, salary, manager and department. It will be referred to in the rest of the paragraph.

End of Example

There are two types of relations:

1) base relations, which are physically stored
2) views: virtual relations, which are dynamic windows on the data base.

View tuples are built from the base relation(s) on which they are defined. In general, any query whose result is a relation may be used to define a view. When information in a base relation changes, the information visible through views defined on that relation changes with it.

Something important we have to care about is that granting, checking and revocation of authorization must be dynamic, when tables and views are created and destroyed dynamically.

I.3.1. Authorization on tables

First a description of an authorization mechanism which applies to all objects (e.g. views and relations), follows. Objects will be referred to by the collective name tables. There is no central data base administrator in the usual sense of the term. Any data base user may be authorized to create a new table. Doing so, he is fully and solely authorized to perform actions upon it.

The following method, devised by [Grif et al.76] and [Lindsay et al.79], applies to System R and the SEQUEL language.

The GRANT command

The persons involved in this command are called grantor (the person who gives) and grantee (the receiver) respectively.
When the creator of a table wants to share his table with other users, he may use a GRANT command to give various privileges on that table to individual users. Among those privileges are:

- READ , to read tuples from the table
- INSERT, to insert new tuples into the table
- DELETE, to delete tuples from the table
- UPDATE, to modify data in the table

Example I.3.2

GRANT READ, INSERT ON EMPLOYEE TO B. After the grant B posseses the read and insert privileges on the EMPLOYEE relation.

End of Example

There is also a GRANT option, which permits the grantee to further grant his acquired rights to other users.

The REVOKE command

The persons involved in this command are called revoker (the person who revokes) and revokee (the person who loses a privilege) respectively.
Any user who has granted a privilege may withdraw it afterwards by issueing the REVOKE command. Thus the authorization system must retain pairs (grantor, grant), to insure that a grantor can revoke only those privileges which he previously granted. Furthermore, upon revocation the system must (recursively) revoke authorization from those to whom the grantee granted privileges for the table.
Let G1, G2,..., Gi-1, Gi, Gi+1,....,Gn be the sequence of grants of a certain privilege on a given table by any user before any REVOKE command has been given. If $i < j$ then Gi occurred before Gj.
Suppose Gi is revoked. Thus the sequence becomes
G1, G2,...Gi-1, Gi, Gi+1,...Gn, Ri,
which is formally defined to be semantically equivalent to
G1, G2,...Gi-1, Gi+1,...Gn, e.g. as if Gi has never occurred.

Consider the following sequence of grants, where user A is the creator of the relation:

A : GRANT INSERT ON EMPLOYEE TO B WITH GRANT OPTION
B : GRANT INSERT ON EMPLOYEE TO C
(*) A : REVOKE INSERT ON EMPLOYEE FROM B ,

which, according to the semantics of REVOKE, is equivalent to

(**) B : GRANT INSERT ON EMPLOYEE TO C.

This fails however, since B does not have the INSERT privilege on EMPLOYEE. As described above the revokee's(B's) grants of the revoked right (to C) must also be revoked, when issueing command (*). After that grant (**) will no longer exist.
Still the decision about exactly which privileges are to be revoked is not obvious. One might expect that if the revokee possesses another grant of the revoked right, i.e. the same grant from another grantor, then recursive revocation should not take place.

Definition I.3.3

A grant graph is a graph in which

1) the nodes represent users
2) an arrow on an edge from X to Y indicates a GRANT from X to Y. The edges are marked with the intended privileges. No mark means that all privileges are granted.

A grant chain is a subsequence A1,...Am of the nodes of a grant graph in which each Ai grants the indicated privileges to Ai+1 (1 <= i <= m-1) .
End of Definition

Consider the following grant graphs:

Figure I.3.1

Suppose in both cases the grant from B to X is revoked. In both cases there is another incoming grant for X (from C and D respectively). However, in the right graph this incoming grant is the last component of a cycle in the chain following the revokee. This cycle has no longer right to exist upon the mentioned revocation.
So a revocation algorithm must distinguish between the case where the second incoming grant is part of a cycle originating at the revokee (in which case it must also be revoked) versus the case where the grant stems from the table creator on a path which does not go through the revoker (in which case it is allowed to remain).

To achieve this the algorithm traces the grant chains from the revokee back to the creator of the table. If every such path passes through the revoker (as is the case in the right graph in figure I.3.1) the revokee's privilege should be recursively revoked. If there is a path back to the creator which does not pass through the revoker (as is the case in the other graph) then the revokee should retain the privilege.
To decide whether to revoke recursively or not without tracing the grant graph, we use authorization tuples of the form
(TABLE, GRANTEE, GRANTOR, READ, INSERT, DELETE). Here the TABLE element contains the tablename, the GRANTEE and GRANTOR elements contain the usernames of the grantee and grantor respectively. The last three elements are timestamps each indicating the relative time of the corresponding privilege. Timestamps are initialized on 0, and for each grantor's commands they are monotonically increasing. Besides no two GRANTs of different users are tagged with the same timestamp.

Example I.3.3
Assume that after a sequence of events SYSAUTH, the table of authorization tuples, looks like:

TABLE	GRANTEE	GRANTOR	READ	INSERT	DELETE
EMPLOYEE	X	A	10	10	0
EMPLOYEE	X	B	20	0	20
EMPLOYEE	Y	X	30	30	30
EMPLOYEE	X	C	40	0	40

Suppose at time t = 50, B issues the command
REVOKE ALL RIGHTS ON EMPLOYEE FROM X. Clearly, all nonzero entries in the last three elements of the second tuple of SYSAUTH must be removed (set back to 0). In order to determine which of X's grants should be revoked we

form:

- a list of X's remaining incoming grants:

TABLE	GRANTEE	READ	INSERT	DELETE
EMPLOYEE	X	(10,40)	(10)	(40)

- a list of X's grants to others:

TABLE	GRANTEE	READ	INSERT	DELETE
EMPLOYEE	Y	(30)	(30)	(30)

The DELETE grant by X at time t = 30 must be revoked, since his earliest remaining DELETE privilege was received at t = 40. However, X's grants of read and INSERT at t = 30 may remain, because there is still an incoming grant at an earlier time for them.

End of Example

In a sort of Pascal the revocation algorithm becomes:

```
type table = tablename;
     grantee = username;
     grantor = username;
     privilege = (READ, INSERT, DELETE);

procedure REVOKE (t:table, x:grantee, y:grantor, r:privilege);
   begin

 (* delete in table t x's authorization for r obtained from y*)
 entry for r := 0 in the tuple (t, x, y, .., .., ..) in SYSAUTH;
 (* find minimum timestamp for the grantee's remaining grantable r on t *)
 m := current timestamp ;
     for each grantor u such that the tuple ≅ 1∓
         (t, x, u, .., .., ..) is in SYSAUTH
      do if 0  < ≅ 17'entry for r < m
         then m := s entry for r;
 (* revoke x's grants of r on t which were made before
    time m *)
     for each grantee v such that the tuple  1∓
         (t, v, x, .., .., ..) is in SYSAUTH
      do if 0 <  17'entry for r < m
         then REVOKE (t, v, x, r)

   end
```

Proof: Immediate from the solution indicated just before example I.3.3.

I.3.2. Authorization on views

The DBMS provides the ability to define views on top of the base relations, and on top of other views. The view-definition permits data value-dependent access to a table. By granting access to a view in order to grant access to selected rows and columns subsets or data dependent access, the grantee may query that view without necessarily having access to all the underlying tables. He may also issue INSERT, DELETE, and UPDATE commands against the view subject to his granted privileges on that view and to the semantics of data modification through views.

When a user A defines a view V, he is the only one who is authorized to perform actions upon it. However, he is not fully authorized, because

1) certain operations (e.g. creation of an index) may not be performed on any view. Others may not be performed on certain views.

2) user's authorization on a view must be restricted to the set of privileges of the user on the underlying table(s). Besides the following two statements hold:
   - when a view involves more than one underlying table then
     { A's privileges on V } = $\cap$ A's privileges on T's
   - privilege P on V is grantable iff A holds a grantable P on all underlying tables of V.

The authorization tuple of V contains A's privileges and the time of definition.

The GRANT command

Views may be granted to other users and, as is the case for base relations, we have the identity:

{ grantable privileges } = $\cup$ grantable privileges from G's

The REVOKE and DROP commands

Revoking is even more difficult than with tables: at REVOKE time we need not only to recursively revoke grants of a privilege or object, but also to drop (remove the definition) or re-authorize (computing the remaining privileges) views built on that object. Again, a sequence of grants and view definitions followed by a revocation or drop, is formally defined to be semantically the same as if the grant or view definition had never occurred.

Let us momentarily restrict the authorization mechanism in the following way:
- each user has either all privileges (and can grant them) or none
- a user may possess only one grant of any given table.

Then the set of grants and view definitions based on a table can be represented by a tree in which the nodes are (user, table) pairs and the arcs are grant and view definitions. REVOKE cuts a grant arc and DROP cuts a set of view definition arcs. Both also prune the entire subtree originating at the removed edge(s).

Let SYSAUTH again denote the table of authorization tuples, which are now of the form (TABLE, GRANTEE, GRANTOR).

Let SYSVIEW denote the system table containing the view definition tuples. A tuple

(UNDERLYING TABLE, VIEW, DEFINER) reflects the fact that the user DEFINER used the UNDERLYING TABLE in defining the VIEW. We can now write two similar recursive procedures in a sort of Pascal to implement REVOKE and DROP:

```
type table = tablename;
     view = viewname;
     grantee = username;
     grantor = username;

procedure REVOKE ( t:table, x:grantee, y:grantor );
   begin
     delete the SYSAUTH tuple for the (t, x, y) grant;
     for each grantee u such that (t, u, x) is in SYSAUTH
      do REVOKE (t, u, x);
     for each view v such that (t, v, x) is in SYSVIEW
      do DROP (v)
   end ;

procedure DROP (v:view);
   begin
     delete v's definition from the system;
     for all users u1 and u2 such that (v, u1, u2) is in SYSAUTH
      do REVOKE (v, u1, u2);
```

```
    for each view f and user u such that (v, f, u) is in SYSVIEW
     do DROP (f)
  end.
```

The extension needed to generalise this authorization mechanism is the use of timestamps, analogous to the preceding section. So those grants are revoked which

- grant a dropped view
- are not supported by remaining earlier incoming grants.

Those views are dropped which

- are built using a dropped view
- are built on a granted table, not received before view definition time.

General selective granting or revocation and re-authorization is just like before.

I.3.3. Authorization of application programs

A USERID can be used to authorize a user who wants to run an application program. The two stages of such a program, installation and invocation, may be under the control of different users, with different authorization privileges. This justifies defining a new object 'application program' in the DBMS. It has the special property that its access controls are separable from the privileges utilized within it.
Installing such a program, which is done by the application programmer, consists of entering it in one of the DBMS catalogs, storing the program in a safe place and authorizing one or more end users (see page -i-) to use it. The installation component of the DBMS checks the installer's authorization against the set of DBMS requests in the program. Installation happens iff the installer is authorized to perform each request . After that the installer obtains the RUN privilege, the ability to invoke the program. This is the only privilege for application programs.
After installing the program the installer may GRANT the RUN privilege further iff he was authorized to GRANT all the privileges needed to perform the DBMS requests in the program.
When installing a program a record is kept of its table references and the installer's authorization on those tables, to be able to invalidate the program when any of these tables is dropped, or when a (some) privilege(s) on those tables from the installer is (are) revoked. Invalidation eventually results in revoking all the RUN privileges of the program.

The main reason for defining application programs is to encapsulate objects so that another user may use them without violating their integrity. Similarly, we defined views in order to be able to authorize (value-dependent) portions of data without fully authorized use of the underlying tables. Thus in authorization checking for an application program there is a dichotomy of requests which appear explicitly in the program versus those which are input upon execution time and are passed through by the program to the DBMS.
If an application program references tables at two or more sites it is not clear which site should store the program or how revocation/invalidation should work. One possibility could be to store it at only one site. In this case remote accesses are needed and the dependencies on local objects must be reflected in each site where the program is stored. The other extreme would be to store it everywhere, which implies a replicated file problem (see chapter II.6.2.1. on multiple copy update). In finding a compromise a conflict arises between doing authorization checking at every access (in order to maintain site autonomy) and the described way of doing it early (for efficiency of execution). A deliberate choice of where to store the program helps to resolve this conflict.

I.4. The hierarchical database case

An important implementation of an authorization mechanism in a hierarchical database system is the SAGE system described in [AdAz 85]: a GEneralized Authorization System with the following properties:

- it intends to protect complex objects (e.g. texts, graphics and documents)
- as was the case in the relational case, the authorization function is decentralized among the users
- the organizational structure where the users are working is modelled as a hierarchy.

Compared with the system of Chapter I.3 the system is favourable in several ways:

1) there is no total decentralization in the following sense: instead of each user managing his own objects and dynamically granting privileges to or revoking them from other users, any object is owned by a unique user at a given time but this ownership can be transferred later to another user
2) no complex propagation or revocation algorithms are needed because the hierarchy provides implicit privileges on objects
3) the real life organization in which the DBMS involves is now considered: the user is associated with a particular position he occupies in the organization (e.g. an office environment).

The considered network architecture consists of a Generalized Database Server (GDS) and a set of workstations (WS) connected through a local network:

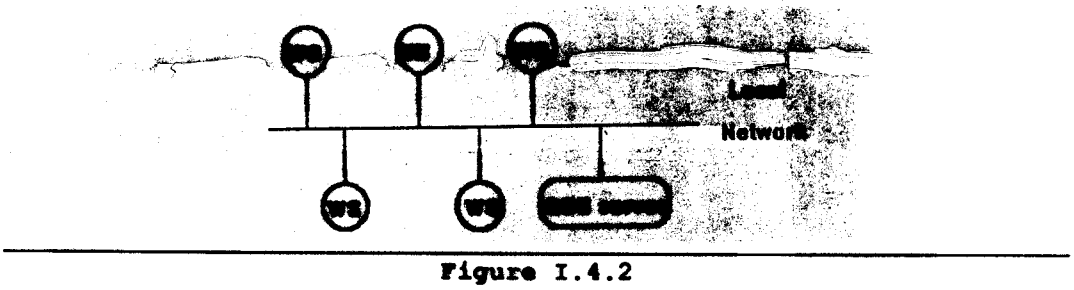

Figure I.4.2

Each WS provides its local set of users some desired local computing facilities.Through the network these users share the database stored at and managed by the GDS. For this database the authorization mechanism is intended.

I.4.1. User management

Assume that a typical user is an employee of the enterprise using the system.
The hierarchical enterprise organization can be represented by a tree, where each node corresponds to a specific role played by an employee in the enterprise. Each user at a given node creates and manages his own data. By default all the superiors of a user can access in read mode the data created by that user, but the user is able to deny such an access. The root user plays the role of Organization Manager (OM). Another special role is provided by the database administrator (DBA), who has the following special rights:

- tree management:
  1) user creation or deletion
  2) user identification
- operations on groups of users:
  1) creation

2) deletion
3) adding of a user
4) removing a user

This role is held by only one user at a given time.

Each user has a code which corresponds to his position in the database. The tree might be imagined then as a catalog where each row describes a user by his

- name
- code
- type (group or single user)
- position
- child(ren), the first dependent(s) in the tree
- next (adres of next code of the user, if any)
- sibling
- object creation privilege(s)
- DBA privilege.

It will be coded to simplify privilege management.
It might be advantageous to grant privileges to a group of users. There are two types of groups:

- those implicitly defined by their level or their ancestors in the tree
- those explicitly defined by the DBA.

I.4.2. Objects to be protected

It is a well known fact that the semantics provided by the relational model are rather poor. Therefore now a data model based upon entity-relationship and abstract data type mechanisms is used. It has two concepts: entity and association. Both have their usual meaning (see page -ii-) and are described using data types similar to the Pascal type mechanism. The attributes of an entity may be elementary or complex (e.g. documents, a concept defined to handle complex objects like text and images, letters or reports).
An object is any data occurrence handled by the GDS. One can distinguish between:

- simple objects (an entity with simple attributes or an associaton occurrence)
- complex objects (an entity with simple and complex attributes)

For any object the privacy constraints may be considered as semantic information associated with it.

I.4.3. Privilege management

I.4.3.1. Privileges on objects

We have the usual operations SELECT, INSERT, DELETE and UPDATE and their corresponding privileges on a set of objects. Besides there are some special privileges:

- the DBA privilege (which does not imply any access rights on objects)
- the FORBID privilege which allows a user to "hide" some of his objects from his superiors in the hierarchy
- the CREATION privilege which may be granted and revoked only by the DBA, and which permits a user to define new objects.

Predicates can be used to restrict the right that a user possesses on an object.
Example I.4.4
For the entity EMPLOYEE defined as
type EMPLOYEE : entity

```
            key num     : integer end key
              name    : string
              age     : integer
              salary  : real
              address: address
              job     : string
              departm: string
    end
```

one could restrict the access right for a certain user to the employees of a certain department.
End of Example

User privileges on objects are represented in the catalog PRIVILEGES. Each line in this catalog gives the user's code, his type, the object owner, the object type and the possible privileges. In case of restricted access the privilege columns also contain pointers to a table where the description of the restriction is stored.
If a level of the tree is protected with some predicates then they apply to all lower levels.
If a level is protected with a set of predicates belonging to various upper levels then the access to this level must satisfy the predicate composed of the intersection of these upper level predicates.

When a simple GDS-object is accessed from a workstation WS its protection is only accomplished in the server. If it concerns a document, protection is also at the WS-level: the document is transferred to WS simultaneously with the user's access privileges on it.

I.4.3.2. Granting

The tree structure of the enterprise induces two privileges:

1) all ancestors of a user in the tree acquire the read privilege on all objects created by the user. However, with the FORBID command the user can deny such an implicit access. To insure a certain degree of user's responsibility, the UPDATE or DELETE privilege of a document cannot be granted implicitly.
2) when a group receives a privilege on an object it is implicitly granted to all users in the group.

Furthermore there is the possibility to explicitly grant a privilege.

The owner of an object is the only user who is allowed to grant access privileges on that object to other users. He may also "grant" the ownership privilege which is in fact a transfer so that only one user at a time has this privilege.

I.4.3.3. Revocation

This is performed explicitly by the owner of the object. It is simple because privileges on an object are only granted by the object's owner and so the revocation of a privilege from a user won't have any effect on other users. This one-step-operation is much easier than the mechanism for relational databases.

I.4.4. Reorganization of the hierarchy

This authorization mechanism appears to be flexible when modifying the organization:

- arrival of an employee involves inserting a new node in the tree, to which a code is assigned and privileges can be granted in the usual way
- when an employee leaves and his job is deleted , his corresponding code and all granted privileges are deleted. When someone takes over the job the code remains the same (it represents a position) and thus the new employee has all the privileges of his predecessor
- insertion of a subtree can be treated as successive creation of levels
- a complete tree reorganization implies the change of all the user codes.

I.5. Some special purpose authorization mechanisms

In the IMS system authorization is provided at two levels:

1) A segment can be declared sensitive or non-sensitive with respect to a program. Only sensitive segments are accessible to a program.
2) Processing options can be defined on a sensitive segment to determine all possible operations on it.

As to databases following the network approach, for the CODASYL-DBTG system, they have an authorization mechanism which provides security at two levels as well: the first consists of defining sub-schemas, each of which describes all the information accessible to users associated with it. At the second level, the security requirements are specified by the lock and key mechanisms: a user can access data if his key equals the lock defined in the schema. See e.g. [Date 81] for further details.

CHAPTER II

TRANSACTION MANAGEMENT

II.1. Preliminary remarks and definitions

Transaction management is a component of the DBMS which accesses and controls the execution of the database actions comprising a transaction. It concerns how to give the database user the ability to define transactions as logical units of work which

1) preserve the atomicity property
2) are recoverable i.e. effects of incomplete transactions can be undone.

Transaction management falls apart in three areas:

- transaction initiation and termination, which means identifying the comprising actions and unambiguously committing or aborting all the results of a transaction
- concurrency control: synchronizing accesses to entities in order to control interactions between different transactions in such a way that the database is kept in a consistent state
- recovery: insuring the database's integrity despite of hardware, software, communication or storage media failures and interrupted/aborted transactions.

The transaction management system is also responsible for scheduling system activity and managing physical resources .
Usually its functions are not available from the basic operating system. They are obtained by either extending the operating system objects (e.g. transactions to have recovery) or by providing entirely new facilities (so called independent recovery).
The life of a transaction instance consists of several phases:

1) a BEGIN TRANSACTION action, which sets it up as a recovery unit in the sense of definition 3 on page -ii-.
2) a series of actions against the system state
3) either a COMMIT TRANSACTION action which causes the outputs of the transaction to be made public or
   an ABORT TRANSACTION action which withdraws all actions performed by the transaction, if the transaction runs into trouble.

A COMMIT action must insure that the transaction is successful at all sites.
An ABORT action must insure that there are no side effects of the transaction anywhere.

The DBMS memory is considered to be divided into:

- volatile storage: main memory and virtual memory
- non-volatile on-line storage: disks, both for the database and for the log (where updates are stored)
- non-volatile off-line storage: tape, archive etc.

We assume that the failure modes of these types of storage are independent, e.g. no two of them will fail at a time.
The data in the distributed database may be stored in several ways:

- strictly partitioned: no duplicate objects of entities are provided

- fully replicated: only copies of the complete database are distributed
- partially replicated: the database is partitioned with certain parts being stored as distributed copies.

This chapter is an extensive description of the functions provided by the transaction management component of a distributed database system.

II.2. Initiation and preparation for remote accesses

In order to be able to maintain isolation between concurrent transactions and to guarantee that all effects of a committed transaction will be reflected in the database, each action must be associated with a transaction. Further all actions of a transaction need to be associated.
A new transaction is created when an application program defines a sequence of actions and asks the local DBMS to start a transaction, consisting of exactly these actions. This involves creating a recovery unit in the sense of section II.1 and indicating which actions can be executed concurrently. The initiating site is called the site of origin of the transaction. Its local DBMS assigns a globally unique transaction-id to the transaction.
After the transaction has been created at its site of origin, it can perform accesses to the local database and also to remote data (data stored at other sites). Once such a remote access is required the local database manager sends a work request describing the remote actions required.
Let L denote the list of sites involved in the transaction. The local DBMS of each site in L maintains a transaction descriptor TD kept in volatile storage, which contains:

- the transaction-id
- the transaction recoverable state (to which the transaction will return in case of restart recovery, e.g. because of hardware or software failures)
- a timer which provides monotonic timestamps (in fact sequence numbers) for messages
- a low water mark (LWM): the timestamp of the first received work request for the transaction at this site (zero at the site of origin)
- a list of remote sites accessed in behalf of this transaction along with the LWMs for these sites. Each newly accessed site will be inserted in this list.

For the site of origin the local TD will be allocated and initialized upon transaction initiation. For the other sites in L this happens upon the first received work request.
The recoverable state is irrelevant before creation, during execution and after completion of the transaction. It is only relevant during multi-site transaction commit which will be considered in chapter II.5.

Before a transaction accesses an entity, the entity is locked. The way of locking depends on whether the entity is requested in share mode or exclusive mode. For locking protocols see chapter II.6.1.2.

II.3. Migration

To access a remote site R a transaction must migrate to R. After receiving the associate work request R performs the requested work and sends the answer back to the requesting site.
Setting up such a conversation requires mutual authentication and protection of the messages so that eavesdropping is useless and undetected alteration is impossible (see [PlvL 85a,b]).
When a remote site receives a work request, it checks if the transaction is

already active at the site. If not, a recovery unit is set up, the actions executable concurrently are indicated and the transaction descriptor is initialized. If the transaction is already active at the site, only the timestamp of the answer needs to be updated.
After sending a work request the transaction at the sending site waits for the answer. To avoid the duplicate problem (e.g. the transaction repeats a work request before a delayed answer arrives, causing a pending work request), we assume that the transaction waits for an answer before sending a new work request. Only the last site visited is allowed to generate new work requests. This avoids parallel work requests which could cause transaction indeterminacy. We can relax this restriction if no two work requests will obstruct the serializability of the transaction schedule (see chapter II.6).
The transaction management facility should not place any constraints upon how a transaction migrates through sites of the distributed database, and a site waiting for an answer should be able to process new work requests for other transactions.
Automatic transaction initialization upon receipt of a work request gives rise to a problem in case a site aborts a transaction between the receipt of two work requests. Then the second work request would re-establish the transaction, which is undesirable as the transaction has been aborted previously. As a solution this request must be notified of the abort decision, so that it can complain to the requesting site. The local low water marks (LWMs) and monotonic timestamps help to solve this problem:
Suppose a remote site R answers a work request. R keeps a list L of sites used by it for this transaction in its TD. When answering a work request L is included in the answer. The requestor compares L with its local list L'. Each element missing in L' is inserted. If a site is already in L' the LWMs are compared. When they differ the site has aborted and restarted (see next chapter) and so the entire transaction must be aborted.

II.4. Recovery management

First of all this facility must be able to undo a committed transaction: it should be able to run a new transaction which corrects/compensates the errors of its predecessor.
Instead of committing, a transaction can be aborted. When this happens, all of the transaction's updates must be undone. Transaction abort is invoked to recover from communication failures and resource limitations, e.g. a request for an entity cannot be executed because the node storing the entity is down or the entity is locked on and on by other transactions.
Both the application program (user) and the database manager are allowed to undo the effects of the uncompleted transaction(s). The first is called suicide , the second is called murder. Unless concurrency controls are used to isolate transactions from each other, undoing one transaction may require undoing others.
Considering an abort the concurrency anomalies which will be discussed in chapter II.6 also occur.
Example II.4.1

> Let T1 and T2 be two transactions. Suppose T1 modifies a value which is in turn modified by T2. If T1 is subsequently aborted and T1's value is restored then T2's update is lost.

End of Example

In general, if lost updates and/or dirty reads could occur then aborting one transaction might invalidate the result of other (uncompleted) transactions. So recovery management must be able to remove (undo) the effects of a not yet committed transaction.

On the other hand, if a transaction has been committed, recovery management must insure that all its effects are retained despite the following failures:

1) hardware and software failures which cause the DBMS at one site to be stopped and restarted
2) storage media failures which hamper data retrieval from the on-line database storage
3) failures in communication between different sites. These include:
   - delivery of a garbled message to the right recipient
   - delivery of a message to the wrong recipient
   - loss of a message.
   - a response from the recipient is not forthcoming, causing a dilemma: either retransmit the message or abort the transaction.
   - change of the ordering of messages

To achieve reliable distribution we require that every message is acknowledged by the recipient.

The following paragraphs deal with the recovery facilities implemented at each site to withstand these various kinds of failures. It turns out that all these facilities can be designed and described in terms of three recovery activities:
- transaction abort
- site restart
- storage media recovery.

In general all three forms of recovery will rely upon redundant representation of the database state.

II.4.1. The log

This is a journal file maintained as an alternative representation of the database. Logically, it is a sequential file of records which documents all changes to the database state. However, the log can also be viewed as the real database because it contains the history of all modifications ever made, the on-line data pages being an optimized database allowing rapid access to the most recent database state.
Log records document all recovery relevant database events:
- updates
- transaction termination (both commit and abort)

Besides, recovery management makes various notes in the log to facilitate recovery (such as checkpoint-records, which will be considered later).
The log has two components: one in volatile memory, called the buffer pool, the other in non-volatile (stable) storage, which may be copied for reliability. The buffer pool is a bounded queue; a demanded page will be stored in any empty buffer slot. We assume that the log never fails (e.g. the duplexing is on devices which never fail at a time).
A log record can be written in two ways:
- synchronously, called forcing a log record; this record and all preceding ones are immediately moved from the buffer pool to stable storage. The log writer is not allowed to continue execution until this operation has completed. If a site crashes, the volatile storage is lost. Upon a site restart after a force write, the force-record and the ones preceding it will be available for recovery.
- asynchronously: the record is written in the buffer pool, and is allowed to migrate to stable storage later on (forced or because of a filled up buffer). After a crash the record may not be saved.

Of course the first method requires more time of the log writer, but it may be necessary in order to make the write action recoverable.
For a description of who writes what kind of log records and when, see section II.4.1.3.

Log records can contain information to

- redo the logged action (new values or inserted entities are described). Logs containing only redo records are called forward.
- undo the logged action (the values or entities to destroy are described). Logs containing only undo records are called backward.
- both redo and undo the logged action. Logs containing this kind of records are called bi-directional.

II.4.1.1. Log management

In general data is transferred between volatile and non-volatile storage in pages containing representations of entities.
When an entity is updated, created or deleted the page(s) containing the entity representation are read in the buffer pool, modified in there, and copied back to the disks either replacing its earlier version or to a new location, leaving the old for recovery purposes. One could also think of updating a so called shadow page instead of the original one and the shadow page eventually replacing the original one.
Of these we chose to write pages back to their original position.
In case of updating, inserting or deleting an entity a record which reports on the entity change is added to the log file. This record either contains the physical changes made to specific pages or the logical database operations performed. Physical logging records the file addresses of and changes to all modified pages (e.g. the receiving page as well as index or hash table pages to support access to the new record). Logical logging of a record insertion would describe the file, record type and field values of the new record. Physical logging tends to be more expensive but it leads to simpler recovery procedures.

II.4.1.2. How the log helps in recovery

Consider a site restart. Then transaction abort uses a backward log to undo the effect of uncommitted transactions (e.g. modified buffer pool pages have replaced original pages on disk before the transaction commits). Forward logging allows to apply the effect of committed transactions to non-volatile storage (e.g. non-volatile pages not copied yet from the buffer pool when the transaction commits).
To achieve this, relevant redo-records plus the commit-record are forced on the log upon transaction commit. Together these actions yield a transaction consistent state, to be defined precisely in chapter II.6.

Now consider storage media failures.
Here again a forward log helps in recovery: a damaged page can be made up to date by applying the relevant redo-records of committed transactions.

II.4.1.2.1. Optimalizations

To facilitate transaction abort, a backward log is stored on a direct access device. In doing this for a forward log there is a trade-off: either avoidance of tape handling or good long term integrity plus high capacity for recovery.
If the on-line restart and abort log are stored in direct access storage, they must be confined to a bounded buffer. A log record at the tail of the buffer pool can be overwritten as soon as these two conditions hold:

- if it is an undo-record it is no longer needed for abort and not needed for site restart.
- if it is a redo-record it is not needed for site restart or storage media recovery.

However, there must always be enough log space to perform logging required during site restart plus to take system checkpoints (to be considered in II.4.2.).

II.4.1.3. A log protocol

Assume that each byte of a log record is numbered and that a log record is identified by the log sequence number (LSN), the number of its last byte.
Log records are appended to the log by copying them into the non-volatile buffer pool. A buffer is written to a non-volatile log-file when it is filled up or a when a forced write occurs.
Every page update is described by a physical, bi-directional modify-record, which identifies the updated page and describes the old and new page states. Also, all the log records of a transaction are linked together and contain the transaction-id. Each database page contains a page-LSN equal to the LSN of the last modify log record. We require that all pages modified by a transaction be locked exclusively before being updated and that the locks be held until the end of the transaction. Thus only one incomplete transaction can be updating a page. For more details on locks see chapter II.6.
If in any of the following 5 sections an indication fails of who performs a particular action, then it is performed by the local DBMS of the site at which the command mentioned in the title of the section is to be executed.

II.4.1.3.1. Page fetch

Whenever a database page must be read into the buffer pool, the buffer pool manager selects an empty buffer slot and then reads the demanded page into the buffer. If the page is unreadable, storage media recovery is required. Otherwise, the buffer pool manager adds a fetch-record to the log which contains the page address of the page just read. The LSN of this record is called the fetch-LSN of the record. It gives a lower bound for the redo-records which may have to be applied to the page during restart.

II.4.1.3.2. Page update

Before updating a page, the page must be locked exclusively and pinned in the buffer pool: it is held in the buffer pool and will be fetched if it is not already present. Before unpinning a (modified) page (allowing it to leave the buffer pool), a modify-record is added to the log and the LSN of the modify-record is recorded in the page's page-LSN.

II.4.1.3.3. Page write to non-volatile storage

Allowed unpinned pages (those of which an update has completed, the log records for all updates have been inserted, and the page-LSN has been updated) can be written to disk.
Suppose an object is recorded in non-volatile storage before the log records for the object are recorded in the non-volatile storage log. Then upon a crash the update cannot be undone.
Similarly, if the new object is one of a set which is committed as a whole, then upon a storage media error on the object no consistent version of the set of objects can be constructed from their non-volatile version.
These examples indicate that the log should be written to non-volatile storage before the object itself is written. This protocol for synchronizing logging and updates to non-volatile storage is called the Write Ahead Log protocol. It allows both uncommitted updates to replace previous values and updated pages to be written before the end of a transaction. It increases flexibility in page migration and avoids extra I/O at the end of the transaction.
The buffer pool manager records page write completions by adding an end-write record which contains the page address to the log. Together with fetch-records end-write-records are used to determine which pages (may) have been in the buffer pool at crash time.

II.4.1.3.4. Transaction commit
This action adds a commit-record to the log and then forces this record and all preceding log records to non-volatile storage. All page locks are released once the commit-record has been written. This write is the atomic action which commits all the transaction's updates to the database.

II.4.1.3.5. Transaction abort
Here we ought to avoid having to log undo activity because that might imply redoing an undo.
Abort reads the transaction's modify-records, starting with the most recent one. Remember that all log records of a transaction are linked, so abort does not need to search for them. Each modified page is pinned and the logged update is undone using the old values in the modify-record. Next the page-LSN is reset to the old LSN. After that the page is unpinned. When all the transaction's updates have been undone, an abort-record is forced on the log. Then the transaction's locks can be released.

II.4.2. Checkpoints

From time to time the recovery facility takes a checkpoint so as to limit the amount of log which must be scanned and the number of pages which must be fetched during restart. This involves purging old pages from the buffer pool and recording the currently active transactions and buffer pool contents in a checkpoint-log record. This information is used upon a site restart.
If comparing the fetch-LSN of a page in the buffer pool and the LSN of the checkpoint-log record tells that the fetch was before the last checkpoint, the page is declared dirty and therefore flushed (first pinned in the buffer pool and then written to non-volatile storage). An end-write for the flushed page is logged, then a new fetch-record is logged and the buffer's fetch-LSN is updated. Eventually the page is unpinned.
Fixing a new checkpoint insures that no updates recorded before the penultimate checkpoint will have to be redone or undone during restart.
The checkpoint-record is created after the removal of dirty pages by quiescing the system: new actions are deferred and on-going actions are completed or backed out. At this "action consistent point" the snapshot is taken which results in :

- a list of Active Transactions with the LSN of the first log record of each, from now on denoted by the AT-list.
- a list of Buffer Pool pages with the fetch-LSN of each page, henceforth denoted by the BP-list.

The log address of the checkpoint is recorded in special non-volatile storage called the emergency restart record.

II.4.3. Site restart
This recovery facility consists of three phases:

1) analysis: determine which transactions were incomplete and which pages were in the buffer pool at the time of the crash
2) undo: removes the effects of incompleted and aborted transactions
3) redo: insures that all committed updates are present in the database.

Restart makes three scans over the log each of which corresponds to one of these phases.
[see Figure II.4.3]
This modular design greatly helps in allowing another restart, if a crash occurs before the end of site restart.
In the following description of the phases let mLSN, fLSN, and aLSN denote the LSN of a modify, fetch and abort-record respectively.

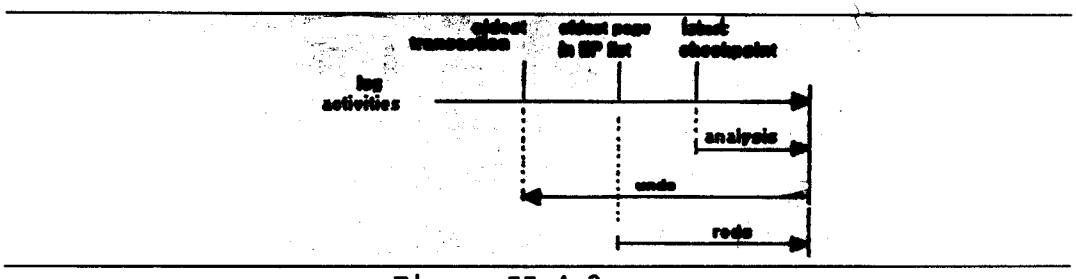

Figure II.4.3

II.4.3.1. Analysis

The information to be collected is stored in the latest checkpoint-record. Its log address is read from the emergency restart record. The AT and BP-lists are created and then the log is scanned forward from the (latest) checkpoint to the end. The various kinds of records are processed as follows:

fetch: if the page is not in the BP-list then it is, together with the LSN of the record, added in there. No modify-records with LSNs less than the fLSN will have to be redone, as promised in paragraph II.4.1.3.1.

end-write: if the page is in the BP-list, remove it. No earlier committed (aborted) updates will have to be redone (undone).

modify: if the transaction is not in the AT-list, it is added in there so that incomplete transactions can be identified upon a redo.

commit/abort:
the transaction is removed from the AT-list. Only pages in the buffer pool may have to be redone/undone.

When the end of the log is reached, the AT-list contains exactly those transactions which had not forced a commit or abort-record before the crash. The BP-list will contain the pages which were in the buffer pool at crash time or had been written before the end-write-record had been written.

II.4.3.2. Undo

Apart from the task sketched above undo prepares for the redo phase: when reading the log forward during analysis, it is unclear whether modify-records need to be applied because the decisive commit or abort-record comes later in the log. During its backward scan undo identifies aborted updates.
Restart undo begins scanning backwards from the end of the log. It begins by making a copy C of the AT-list and then marks each member of the original AT as 'incomplete'. Besides all of them are given abort-LSNs equal to the end of the log.
Only abort and modify-records are relevant in this phase.
They are treated like this:

abort: if the aLSN is greater than the minimum fLSN in the BP-list then the transaction is added to the the AT-list and marked 'complete'.

modify:
- If the record is not in the AT-list it is skipped.
- If it is in the AT list and if the transaction is incomplete (e.g. already occurred in the AT-list after analysis) the page must be checked to see if the update must be undone. The page must also be checked if it is in the BP-list anT if the page's fLSN is less than the aLSN. To check a page it is fetched and the page-LSN is compared to the mLSN. If they are equal then the update is undone and the page-LSN is reset to the old LSN from the modify-record.
- If the transaction is in the AT-list and the page is in the BP-list then an aborted interval is added to the BP-list (the interval lasts from the mLSN to the aLSN).

- If it is the first modify-record for the transaction, the latter is removed from the AT-list.

Restart undo will complete with the AT-list empty.

II.4.3.3. Redo

Only pages in the BP-list need to be considered here because they are the only ones from which committed updates may be missing (e.g. the page had been updated but not yet written before the crash).
Only modify-records are processed:

modify: if the page is in the BP-list then check if the mLSN is in one of the page's aborted intervals. If not, compare the mLSN with the fLSN in the BP-list. If the first is greater then fetch the page. If the page-LSN of this page equals the old LSN in the modify-record, then redo the update and set the page-LSN to the mLSN.

When restart reaches the end of the log it is complete.
Restart recovery finishes by adding abort-records to the log for all incomplete transactions (elements of C, the copy of the AT-list). This insures a consistent state of the log. Then two checkpoints are taken. This removes dirty pages and updated pages from the buffer pool so that the system is cleaned up completely and ready for normal processing.

II.4.4. Storage media recovery

This consists of the recovery of damaged database pages. It can be implemented easily if physical logging is used because then it is quickly clear which pages are updated by which log records.
Damaged pages cannot be referenced during storage media recovery, while non-damaged pages can.
A description of the facilities needed to repair damaged pages follows.

II.4.4.1. Image dumps

To restrict the number of log records to be examined in case of storage media failures, image dumps of the complete database are periodically constructed. In this case scanning of records more recent than the image dump suffices.
An image dump is formed by scanning over the pages of the database and making a copy of each page on the dump media. If we take care to make the image dump transaction consistent (to be considered in II.6.) then upon storage media recovery no changes to dumped pages will have to be undone. Such a consistent dump can be obtained by first halting and then restarting the system after a while (see the chapter on checkpoints). If this is impractical, a so called fuzzy dump of individual transaction consistent pages is obtained by locking each page for reading, copying the page to the dump media and then unlocking the page. This insures that no incomplete transaction has modified the page.

II.4.4.2. The archive log

This is a collection of tapes storing log records used to redo updates to image dump pages. If dLSN denotes the LSN at the time the (fuzzy) image dump started, then the archive log has to contain redo-records for all updates by committed transactions logged after dLSN. By eliminating all records from aborted transactions media recovery can be quickened.
The archive log is created by scanning forward the on-line log from dLSN. During that, the AT-list is maintained by processing the following records as indicated:

modify:

- if the current modify record's transaction is not in the AT-list, then scan further to find the associated commit or abort-record. If not found , archive logging must await completion of the transaction. Otherwise, the transaction is entered in the AT-list with an indication of how it terminated.
- if the transaction owning a modify-record is in the AT-list and it has been committed, then redo info from the record is added to the log; if the transaction has been aborted the modify-record is skipped.

commit/abort:

the corresponding transaction is removed from the AT-list.

II.4.4.3. Restoration of a page

First the page is locked exclusively to prevent from any access. Next it is located and read from some image dump (preferably the most recent). Updates to the page are then redone using the archive log tape containing dLSN and all subsequent archive log tapes. Afterwards the on-line log for more recent updates to the damaged page is scanned from the LSN of the last record in the archive log. When the end of the on-line log is reached, the damaged page has been restored to its most recent transaction consistent state.

II.4.5. Extension to distributed recovery

With a few slight extensions the local site recovery described can support recovery for distributed databases. This is described in the next chapter on transaction commit.

II.5. Commit

To be able to value the problem solving aspect of a commit protocol, consider the Generals Paradox:

There are two generals on campaign. Both want to capture a certain hill. If they simultaneously walk on the hill they are assured of success. If only one marches, he will be annihilated. The generals can communicate only via runners who are unreliable in the sense that they may get lost when they venture out of camp.

Problem: Find a protocol which allows the generals to march together even though some messengers get lost.

Proposition II.5.1

No such fixed length protocol exists.

Proof: Let P be the shortest such protocol. Suppose the last messenger in P gets lost. Then either this messenger is useless or one general does not get a needed message. By the minimality of P the message is not useless so one general does not march because of the loss, causing the other to be annihilated. Contradiction.

End of Proposition

There is also the Byzantine Generals problem (see [DeMi et.al. 82] and [DoSt 82]). Here we have n generals, m of which may be faulty. Each general Gi has a private value vi. The goal is to achieve Byzantine Agreement: each non-faulty general Gi sends and receives messages in order to obtain a vector V = (V1, V2,...Vn) of n values such that Vi = vi, and if Gj is another non-faulty general, Gj computes the identical vector under the assumtions that messages are never lost and always delivered intact within a specific time. The faulty generals behave arbitrary in the sense that their behaviour cannot be anticipated.

In both problems it is essential that the (non-faulty) participants eventually have the same information at their disposal before the decision/agreement is made.
What do these problems arising in the area of "defense" have to do with a peaceful transaction commit?

Once a transaction has been initiated it must eventually complete. If all goes well the transaction can commit its updates to the database and notify the user of completion. Committed updates are visible to other transactions and will be recovered if the system or storage fails.
If the transaction has updated database entities at more than one site, we have to take pains to insure that all the sites involved in the transaction commit together which was also the purpose in the preceding generals problems.
Other desirable properties of a commit protocol are:

- the ability to "forget" the outcome after a while
- minimal overhead in terms of log writes and message sending
- optimized performance in the no-failure case
- exploitation of read-only transactions.

There are several protocols to achieve such a uniform commit.

II.5.1. Two-phase commit protocols

This kind of protocols is described for the first time in [Gray 78]. Here one site, called the coordinator makes the final commit/abort decision after all the other sites (the subordinates) in the transaction have already agreed to respect the coordinator's decision. A participating site is allowed to abandon (abort) a transaction at any time up to the moment when it consents in taking part in the commit. This compromises site autonomy but insures that sites will remain able to commit/abort until the coordinator is able to decide.
During the first phase of a protocol of this kind each site of the transaction (they are all stored in the transaction descriptor) is queried by the coordinator if it can commit its part of the involved actions (e.g. if it has not abandoned yet). If it does not need any additional resources each site becomes recoverably prepared to go either way. Then it awaits the coordinator's decision which is made once all sites are prepared to commit.
Then the protocol enters phase two which consists of notifying all sites of the outcome.
To make true the concept 'recoverable state' (which is also kept in the transaction descriptor) it must be stored in non-volatile storage. The local log file is intended for that.

II.5.1.1. The linear two-phase commit protocol

Here a participating site can assume four states:

UNKNOWN: entered when either the site has no change records in the log or all changes have completed and all the site's volatile records of the transaction have been flushed. So the protocol need not distinguish between unbegun and completed transactions.

READY: entered when the site has been asked to prepare for multi-site commit but not yet answered this request. Hereafter the site will abort or commit but not until it is notified of the coordinator's decision. Thus the shared (read-only) resources can be released and updates by other transactions will not influence this state. Furthermore after a failure the site can recover the effects and resource reservations from this state.

COMMITTING: entered when the coordinator's decision has been to commit and the

site receives notice of this fact.
ABORTING: entered when the site is being asked to commit a certain transaction but has abandoned that transaction previously

The sites are ordered into a sequence and the commit proceeds as messages travel from site to site according to the ordering in the sequence.
The protocol proceeds as follows:

1) The site of origin S initiates the commit process by becoming READY. S fabricates a PREPARE-message containing the transaction-id, and the list L of sites used with S at its head. S sends this message to the next site.
2) Any site R receiving a PREPARE-message checks if it has not abandoned the indicated transaction. If it has R returns an ABORT-message to the sender (its predecessor) and enters the state ABORTING. Otherwise the receiver enters the READY-state and sends a PREPARE-message to the next site (if any). R awaits the coordinator's decision and is only allowed to resend a PREPARE if no response is forthcoming.
3) If the last site C in the sequence receives a PREPARE it becomes the coordinator and makes the decision to commit or abort. If the transaction is active (not abondoned) at C, it directly enters the COMMITTING state (bypassing READY). C sends a COMMIT message to its READY predecessor.
4) Any READY site R' <> S which receives a COMMIT message, enters the COMMITTING state. R' sends a commit-ACK to its successor and sends a COMMIT message to its predecessor. If R' receives an ABORT message it acts similarly: the ABORT is acknowledged to its successor and an ABORT message is sent to its predecessor after which it enters the UNKNOWN state.
5) A COMMITTING (ABORTING) site receiving a commit-ACK (abort-ACK) enters the UNKNOWN-state.
6) when S receives the COMMIT it sends a commit-ACK and returns to the UNKNOWN state.

The protocol steps invoke the following storage actions:

1) An in-doubt log record containing L is forced to non-volatile storage.
2) R forces an in-doubt log record (with L) to non-volatile storage.
3) The coordinator C forces a commit-record to the log. This is the atomic action which irrevocably commits the transaction. C releases the transaction's local locks and resources.
4) In case of a COMMIT R' forces a commit-record to the log. Each COMMITTING site must maintain its volatile memory transaction descriptor TD until the COMMIT is acknowledged. In case of an ABORT R' drops the TD.
5) The site concerned logs a done-record and drops its TD.
6) S forces a commit-record. Thereafter S drops all volatile memory records of the transaction.

II.5.1.2. The centralized two-phase commit protocol

Instead of propagating the PREPARE and COMMIT messages from site to site, a single site can send all the PREPAREs and COMMITs. This is done by the site of origin which is the coordinator at the same time. All sites involved in the transaction prepare and commit in parallel.
Apart from the states mentioned in the preceding protocol there is the COLLECTING state, recoverably entered by the coordinator upon starting the commit. Now the COMMITTING and ABORTING state are reserved for the coordinator in case he decides to commit (abort) the transaction respectively.
The protocol proceeds as follows:

1) The coordinator C enters the COLLECTING state and sends a PREPARE-message to every other site in the transaction.

2) When site R receives a PREPARE message and the transaction is still active at R it enters the READY state and sends a vote YES to C. Then it waits for C's response. If R cannot commit it sends a vote NO to C.
3) C collects all votes. If any vote NO arrives then the transaction will be aborted: C enters the ABORTING state and distributes ABORT messages over the other nodes. If all sites responded with a vote YES then C commits the transaction by recoverably entering the COMMITTING-state. It then sends COMMIT messages to the other sites.
4) When a READY site R' receives a COMMIT-message it sends a commit-ACK to C. When it receives an ABORT it sends an abort-ACK to C.
5) C collects all the ACKs and enters the UNKNOWN state.

The following protocol steps invoke storage actions:
1): C forces a collect-record with the list L of sites-used to the log.
3): C forces an abort/commit-record to the log respectively and releases the transaction's local resources.
4): In case of an ABORT, R' forces an abort-record to the log; when R' receives a COMMIT it forces a commit-record to the log and releases all volatile memory transaction records. In both cases R' enters the UNKNOWN state.
5): C releases its volatile memory transaction records and writes an end-record (e.g. "forgets the transaction").

II.5.1.3. Remarks on and comparison of two-phase commit protocols

By requiring the subordinates to send ACKs the coordinator makes sure that they all know of the final outcome.
By forcing their commit/abort-records before sending the ACKs the subordinates make sure that in case of recovery from a failure they will never be required to ask the coordinator about the outcome.

Let N be the number of sites involved in the transaction and let "#" denote: "number of". Then we can fill up the following table:

	linear	centralized
# messages	3(N-1)	4(N-1)
# message delays	2N-1	4
# message delays to commit point	N-1	2
# recoverable state changes	N	2N

Figure II.5.4

The centralized protocol can drop read-only sites from the second phase (they can become COMMITTING immediately) while the linear protocol cannot.
The centralized protocol requires fewer delays when N >= 3. We can decrease total message delay by letting two sites follow the linear protocol and the others the centralized one.

II.5.2. Extensions of the centralized two-phase commit (2P) protocol
In this paragraph 2P will an abbreviation for the centralized two-phase commit protocol, and C will denote the coordinator.
In [MoLi 83] special attention is paid to performance aspects (namely # messages and # written log records) of the commit protocol. In a stepwise fashion two new protocols are derived from 2P: Presumed Abort (PA) and Presumed Commit (PC). They are based on the general principle that a

subordinate does not acknowledge a message before it has made sure (by forcing a log record reflecting the information in the message) that it will never need to ask the coordinator about that information.

First consider the behaviour of the recovery process RP under various failures of 2P.

- If RP finds a transaction T in the COMMITTING (ABORTING) state, it tries to send COMMIT (ABORT) messages to the subordinates that have not yet acknowledged.
- If C notices a failure of a subordinate whose vote he is waiting for, he aborts the transaction.
- If a subordinate S notices a failure of C before it gets into the READY state it unilaterally aborts the transaction. If the failure occurs after S has become READY then RP is employed.
- If RP receives an inquiry message from a READY subordinate it consults the information in virtual storage. If that info says the transaction is in the ABORTING or COMMITTING-state, S is appropriately informed. However, there might be no information at all in the virtual storage, e.g. after this succession of steps:
  1) C sends out a PREPARE to all subordinates.
  2) C crashes before receiving the vote of subordinate S (and so has not decided to commit/abort.
  3) On restart it aborts and does not inform S (because it does not know of its coordinatorship anymore)

  In this case the correct answer for an inquiry of S is an ABORT-message.

A careful examination of this scenario will lead towards the PA protocol.
In the following two paragraphs S denotes a subordinate and RP denotes the recovery process of S.

II.5.2.1. The Presumed Abort protocol (PA)

We can remark that it is safe for C to forget a transaction immediately after it decides to abort instead of waiting to be sure that all the subordinates are aware of the decision to abort. Thus:

- the abort-record need not be forced (neither by C nor by S)
- no ACKs for ABORTs need to be sent by the subordinates
- C neither needs to record the names of the subordinates in the abort-record nor write an end-record after an abort-record
- if C notices the failure of S while attempting to send an ABORT to S he does not need to hand the transaction over to to RP: if RP sends an inquiry it will find out about the abort.

So far PA and 2P have the same performance. However, we can improve on a read-only transaction site R (e.g. one where no updates to the database are performed). If R receives a PREPARE and it discovers not to have done any updates (i.e. no undo/redo log records have been written) then it sends a READ vote, release its locks and forgets about the transaction and so does not write any log records. It does not matter for R whether the transaction will abort or commit, so C does not need to send a COMMIT/ABORT message to R.
Hence:

- for a completely read-only transaction neither C nor any S writes any log records, each S sends 1 message to C, and C sends 1 message to each subordinate.
- for a partially read-only transaction C sends 2 messages (PREPARE and COMMIT) to each S which has updated and 1 message to the others.
- C writes 2 records (a forced commit-record and an unforced end-record) if there is at least one S which has updated, and just 1 record (a forced

commit-record) otherwise.

The name of the protocol stems from the fact that the transaction is presumed to have aborted if there is no informatiion in volatile storage.

II.5.2.2. The Presumed Commit (PC) protocol

The PA approach is in fact rather pessimistic: most transactions are expected to commit, so it is natural to try to eliminate the ACKs for COMMIT-messages. A simplistic idea would be to leave them away indeed and requiring only abort-records to be forced while commit-records need not. Now in the no-information case of the preceding paragraph the transaction is presumed to have committed, hence the name of the protocol.
This extreme simplification fails when C has sent out PREPAREs and crashes before receiving the vote of an S which has entered the READY state: when RP inquires C the answer would be a COMMIT, which is inconsistent.
C can solve this by recording the names of each S before any of them can become READY. Then C's restart process, which carries out the abort, will know whom to inform (and get ACKs from) about that.
These adaptations lead to the Presumed Commit (PC) protocol. C behaves as in PA except:

- before sending the PREPAREs it forces a collecting-record with the names of each S, and enters the COLLECTING-state.
- it forces both commit and abort-records.
- it requires ACKs only for ABORTs.
- it writes an end-record only after an abort-record.
- only when in the ABORTING-state it may (on notifying a crashed S) hand over the transaction to the restart process.
- in case of a complete read-only transaction it writes a commit-record at the end of the first phase in PA and then "forgets" the transaction.

Each S behaves as in PA except that

- now it forces only abort-records
- it acks only ABORT-messages.

II.5.2.3. Comparison of 2P, PA and PC

To be able to compare the performance of 2P, PA and PC we fill up the following table:

Protocol / Type	Coordinator: UPDATE (with update at subordinate)	Coordinator: UPDATE (without update at subordinate)	Coordinator: READ-ONLY	Subordinate: with update	Subordinate: without update
2P	2,1,-,2	–	–	2,2,2,-	–
PA	2,1,1,2	1,1,1,-	0,0,1,-	2,2,2,-	0,0,1,-
PC	2,2,1,2	2,2,1,-	2,1,1,-	2,1,1,-	0,0,1,-

UPDATE: transaction with update
READ-ONLY: transaction without update
An entry k,l,m,n denotes:
k records written, of which
l forced
m for a coordinator: # messages sent to each Read-Only Subordinate
for a subordinate: # messages sent to the Coordinator
n # messages sent to Subordinate with update

Figure II.5.5

which justifies the following conclusions:

- PA is always better than 2P
- PA is better then PC in case of

- completely read-only transactions
- partially read-only transactions in which only the coordinator does any updates.

- depending on the transaction(s) to be run the choice is between PA and PC.

II.6. Concurrency and yet consistency

If transactions run one at a time then each transaction sees a database state which reflects only the effects of completed transactions. Each transaction produces a new database state depending only upon the old database state, the transaction definition and the input. If all transactions were simple (e.g. take in a single message, do something and produce a single message) and the database were stored in primary memory at a single site, there would be no need to interleave the execution of different transactions. However, so called batch transactions (usually not on-line, rather started by a system event and run for a long time in background) and delays inherent to accessing information in secondary storage or at remote sites, require that several transactions execute concurrently in order to improve system response time and throughput as well as resource utilization.
Unrestricted interleaving of database actions from different transactions can cause anomalies which threaten the integrity of the database and compromise transaction atomicity. These anomalies include:

- lost updates: if two transactions read and modify the value of the same record, overlapping execution may cause the record to be modified only once.
- dirty reads: if a transaction T could read outputs from an incomplete (uncommitted) transaction then T sees a state unreachable without concurrency.
- unrepeatable reads: if related data records are read by a transaction T while another one updates them, then T may see inconsistent record values.

Definition II.6.1

Transaction T sees a consistent system state if

a) T does not overwrite dirty (uncommitted) data of other transactions.
b) T does not commit any of its WRITEs until it completes all its WRITEs.
c) T does not read dirty data from other transactions.
d) Other transactions do not update any data read by T before T completes.

End of Definition

Clauses a) and b) insure absence of lost updates.
Clause c) rules out dirty reads.
Clause d) rules out unrepeatable reads.

The system state is not static: it is continually changed by actions performed by processes on the entities. Consistency constraints cannot be enforced at each action: e.g. in moving money from one bank account to another, after the debit action one account will have been debited and the other not yet credited, which obstructs the consistency constraint that the amount of money in the system is constant. This phenomenon of temporary inconsistency is inherent, but the situation that several concurrent transactions end leaving the system in an inconsistent state, called a conflict , is not inherent and

therefore undesirable.
Having grouped actions into transactions , we want to run transactions with maximal concurrency by interleaving actions from several transactions while continuing to give each transaction a consistent view of the system state. The next section deals with various protocols to achieve this so called concurrency transparency. It is assumed that each transaction when executed alone transforms a consistent state into a new consistent state.
Definition II.6.2
A particular sequencing (merging) of all the actions of a set of transactions, which preserves the order within each transaction is called a schedule.
End of Definition

A schedule is called consistent if it gives each transaction a consistent view of the system state.
Observe that not all consistent schedules for a set of transactions give exactly the same state. For example, two consistent seat reservations schemes may differ in the seat assignment. So consistency is a weaker property than determinacy.

II.6.1. Concurrency protocols which preserve consistency

In this section the data is assumed to be strictly partitioned: a given entity is stored only at one node.

II.6.1.1. The partition scheme

To avoid a conflict this technique partitions entities into disjoint classes. One can then schedule transactions concurrently if they use distinct classes of entities.
Unfortunately it is usually impossible to examine a transaction statically to determine which subset of the entities it will use. Therefore we immediately discard this scheme.

II.6.1.2. Locking protocols

A more flexible way to control the interleaving of database accesses from different transactions and to prevent the anomalies mentioned in chapter II.6. is to use dynamic locking: before a transaction accesses an entity, the entity is appropriately locked. This method is described in [Esw et.al. 76], [Gray 78] and [Tra et.al. 79].
There are two kinds of locks:
- LOCK_S: requests the designated entity in share mode. A LOCK_S request is only granted when no other transaction instance is granted the object in exclusive mode.
- LOCK_X: requests the designated entity in exclusive mode. A LOCK_X request is only granted when no other transaction instance is granted the object in any mode.

Further there is the UNLOCK action which releases a lock on a designated object.
When a requested lock on an entity cannot be granted we say that there is a conflicting lock request.
If in the following the kind of lock is not specified, indicated by a plain LOCK action, then the statement holds for both kinds of locks.
Locking is used for several reasons:
1) to prevent a conflict with other transactions by locking out changes made by other transactions and temporarily consistency assertions on the locked entities

2) to enable repeated reads: unless a transaction locks an entity successive reads of the entity may yield distinct values because of an intermediate update
3) as mentioned in the chapter on recovery management locks are of help in recovery and transaction backup.

However, inherent to the concept of lock requests is the threat of both local (within a node) and global (over the system) deadlock. Methods to detect and solve this are described in [Gray 78].
We assume in the sequel that all transactions have the property that they:
- do not relock an entity at step i which is already locked at step i
- do not unlock an entity at step i which is not locked through step i
- end having released all locks.

II.6.1.2.1. General properties of locking

To state and prove some useful results we proceed more formally.

Definition II.6.3
A transaction is a sequence T = ((T, ai, ei))$_{i=1}^{n}$ of n actions where T is the transaction name, ai is the action at step i and ei is the entity acted upon by action i.
End of Definition

Definition II.6.4
A transaction has locked entity e through step i
if j, j <= i, aj = LOCK and ej = e and
k, j < k < i, ak = UNLOCK and ek = e.
End of Definition

Among all kinds of actions there are two generic actions:
- READ: examines but does not alter an entity
- WRITE: alters the value of an entity independent of its prior value.

Definition II.6.5
A transaction T is well-formed if for each step i $\in$ {1,2,..n}
if ai = LOCK then ei is not LOCKed by T through step i-1,
if ai <> LOCK then ei is LOCKed by T through step i
and
at step n only en is still LOCKed by T and an = UNLOCK.
End of Definition

>From the description of the two kinds of locks it follows that transactions are well-formed if READ actions are always preceded by a LOCK_S for the entity, and WRITE actions are always preceded by a LOCK_X for the entity.

Definition II.6.6
Let T be a set of transactions T1, T2, ..., Tm with length l1, l2, .., lm respectively.
Let s = l1 + l2 + .. + lm.
A schedule for T is any sequence of s actions
S = ..., (Tj, ai, ei), ... where i $\in$ {1, 2, ..s} and
j $\in$ {1, 2, ..m},
such that for each k $\in$ {1, 2, ..m}
Tk = { (Tj, ai, ei) | j = k }. The ith action in S, denoted by S(i), indicates that transaction Tj is acting upon entity ei.
Hence S contains nothing more than T1, T2, ..Tm.

The schedule is called serial if for some permutation Π
S = $T\Pi(1)$ o $T\Pi(2)$ o ... o $T\Pi(n)$ where 'o' denotes concatenation.

End of Definition

A serial schedule starts with a consistent state and each transaction transforms a consistent state into a new consistent state. Hence a serial schedule gives each transaction a consistent view of the system state.
As mentioned above, nonserial schedules (those with concurrent transactions) run the risk of giving a transaction an inconsistent view of the system state (an inconsistent set of inputs). Therefore we want the outcome of processing a set of transactions concurrently to be the same as one produced by running these transactions serially in some order.

Definition II.6.7

The dependency relation induced by a schedule S , denoted by DEP(S), is a ternary relation on T x E x T where T denotes the set of transactions in S and E is the set of all entities, defined by

(T1, e, T2) $\in$ DEP(S) iff i,j, i < j, such that
S = ...,(T1, ai, e),...,(T2, aj,e),... with
ai or aj a WRITE action, and
k such that i < k < j and ek = e and ak is a WRITE action.

Informally, if (T1, e, T2) is in DEP(S) then entity e is an output of T1 and an input for T2.

End of Definition

Now we can introduce our intended equivalence notion.

Definition II.6.8

Two schedules S1 and S2 are called equivalent if DEP(S1) = DEP(S2).

End of Definition

Definition II.6.9

A schedule is called serializable if it has an equivalent serial schedule.

End of Definition

Definition II.6.10

A schedule is consistent if it serializable.

End of Definition

So we will use serializability as the consistency preserving criterion. In [Pap 79] it is shown that the question whether a given schedule is serializable is NP-complete. This suggests that most probably there is no efficient algorithm that distinguishes between serializable and non-serializable schedules. But it is also shown that serializability is the weakest criterion for preserving consistency of a concurrent transaction system even if complete syntactic information of the system is available to the concurrency controller so this criterion is satisfying.
If a schedule is consistent then each transaction sees the same state it would see in the corresponding serial schedule. This justifies the dual use of the term consistency to describe states and schedules,
A serial schedule has the effect that all the actions of one transaction execution are completed before the next transaction starts. More formally:

Definition II.6.11

If T1 and T2 are any two transactions out of a serial schedule, and e1 and e2 are any entities, then

(T1, e1, T2) $\in$ DEP(S) $\Leftrightarrow$ (T2, e2, T1) $\notin$ DEP(S) .

End of Definition

We continue to characterize non-serial schedules which are consistent by considering the LOCK and UNLOCK actions of each step.

Definition II.6.12

Entity e is said to be locked by transaction T through step k of schedule S

if j, j <= k | S(j) = (T, LOCK, e), and
j',j < j' < k | S(j') = (T, UNLOCK, e)
End of Definition

Definition II.6.13
A schedule S is called legal if it satisfies the following condition for all k:
if S(k) = (T, a, e) and e is LOCKed by T through step k then there has been no conflicting lock request on e by any other transaction through step k.
End of Definition

Proposition II.6.2
Not every legal schedule is consistent.
Proof: Consider the following schedule for transactions T1 and T2 and entities A and B.

```
T1  LOCK_X A ; T1  A := A + 1 ; T1  UNLOCK A   ; T2  LOCK_X A ;
T2  LOCK_X B ; T2  A := A + 1 ; T2  B := B + 1 ; T2  UNLOCK B ;
T2  UNLOCK A ; T1  LOCK_X B   ; T1  B := B + 1 ; T1  UNLOCK B .
```

Clearly this schedule is legal. But
DEP(S) = {(T1, A, T2), (T2, B, T1)}, so because of definitions II.6.8 , II.6.9 and II.6.10 the schedule is not consistent.
End of Proposition

How to construct a consistent schedule out of a legal schedule?
Lemma II.6.1
Consistency requires that transactions be well-formed.
Proof: Suppose transaction T1 = $(T1, ai, ei)_{i=1}^{n}$ is not well-formed. Then for some step k T1 does not have ek LOCKed through step k. Suppose T1(k) = (T1, WRITE, ek). Define the transaction
T2 := ((T2, LOCK_S, ek), (T2, READ, ek),
(T2, WRITE, ek), (T2, UNLOCK, ek)). Define schedule
S := $(T1(i)_{i=k}$, T2(1) , T2(2) , T1(k) , T2(3) , T2(4) , $T1(i)_{i=k+1}$ Clearly S is legal. But both (T1, ek, T2) and (T2, ek, T1) are in DEP(S) so because of definitions II.6.8 , II.6.9 and II.6.10 S is not consistent. This is intuitively clear: T1 changes ek after T2 reads it but before T2 writes it. Contradiction.
QED

Definition II.6.14
Transaction T = $((T,ai,ei))_{i=1}^{n}$ is called two-phase if for some j < n :

```
i < j   ai <> UNLOCK
i = j   ai = UNLOCK
i > j   ai <> LOCK.
```

Steps 1,2,..j-1 are called the growing phase, in which the transaction is allowed to request new locks. Steps j,..n are called the shrinking phase, in which the transaction may only release locks. After a transaction issues an UNLOCK action, it never issues a LOCK action.
End of Definition

Lemma II.6.2
Consistency requires that transactions be two-phase.
Proof: Consider again the schedule of II.6.2: here T1 is not two-phase because it LOCKs B after releasing A. T2 is well-formed and two-phase. T2 gets the updated A from T1, while T1 gets the updated B from T2. This can't possibly occur in any serial schedule. Hence S is inconsistent.
QED

However, the notions well-formed and two-phase together will guarantee that a

legal schedule is consistent. To prove that we need a lemma.
Let S denote a legal schedule of a set T = {T1,...Tn} of two-phase well-formed transactions.

Lemma II.6.3

The binary relation '<' on T defined by

Ti < Tj iff (Ti,e,Tj) ∈ DEP(S) for some entity e can be extended to a total order << on T.

Proof: Define the integer SHRINK(T) for a well-formed non-null transaction T to be the least integer j such that T UNLOCKs some entity in S(j).

Claim II.6.1

For any transactions T1 and T2 and entity e,
if (T1,e,T2) ∈ DEP(S) then SHRINK(T1) < SHRINK(T2).

Proof: There are integers i,j such that S = ..., (T1, ai, e),..., (T2,aj,e), ... and for all k between i and j , ek <> e. S is legal, so e must be LOCKed only by T1 through step i of S. T2 is well-formed, so e must be LOCKed only by T2 through step j of S. So ai = UNLOCK e and aj = LOCK e. This implies SHRINK(T1) <= i. Since T2 is two-phase, no unlock by T2 precedes step j of S so SHRINK(T2) > j. Therefore SHRINK(T1) <= SHRINK(T2).

End of Claim

Thus we have shown that
if T1 < T2 then SHRINK(T1) < SHRINK(T2).
This defines a total order on T:
T1 << T2 iff SHRINK(T1) < SHRINK(T2).

QED

Theorem II.6.1

S is consistent iff each element of T is well-formed and two-phase.

Proof:

This follows from lemmas II.6.1 and II.6.2

Assume without loss of generality that T1 << T2 ...<< Tn. Use induction on n to show that S is equivalent to the serial schedule T1, T2,..Tn:

n=1: trivial.

IH: Suppose it is true for all k <= n-1. Define the schedule S' := T1((Tj,ai,ei) ∈ S | j <> 1).

Claim II.6.2

S is equivalent with S'.

Proof: (Ta, e, Tb) ∈ DEP(S) iff entity e integers i,j, such that S = ..., (Ta, ai, e)...(Tb, aj, e),... If a <> 1 and b <> 1 then the ordering of these two actions is unchanged, so (Ta, e, Tb) ∈ DEP(S'). If a = 1 and b = 1 the same argument holds. If a = 1 and b <> 1 then action (Ta, ai, e) is probably done earlier than before, but then again the same argument holds. If a <> 1 and b = 1 then in step i only Ta has a LOCK on e because Si legal, and in step j only Tb has a LOCK on e because Tb is well-formed. So meanwhile Ta has UNLOCKed e. This contradicts Ta >> Tb.

End of Claim

By the induction hypothesis $(\,(Ti,ai,ei) \in S \mid Ti <> T1)_{i=1}^{n}$ is equivalent to the serial schedule T2,..Tn. So S' is equivalent to the serial schedule T1,T2,..Tn and therefore consistent. By the claim it follows that S is consistent.

QED

In [Trai et.al 79] the notion of two-phase is strengthened: all LOCKs will be held to the very end of the transaction execution rather than some UNLOCKS occurring prior to that. Thus the necessity property of theorem II.6.1 is sacrificed, but recovery is easier with this newly adopted definition.

For a distributed system the execution of a particular transaction instance is as follows: it begins at some node and then issues successive actions. If the object is local the action is performed locally. If the object is at a remote site the transaction requests the remote node to perform the action on the entity. It then waits for the successful completion of the remote action to arrive before going on with the next action.
The perception of each node is that at any instant it has several actions on local objects to perform. It executes these actions in some serial order called a node schedule. Given a merger of these node schedules one could apply the previous results to discuss consistency. However, since each node is running autonomously it is not obvious that these node schedules can always be merged into a single system schedule.
Theorem II.6.2

> The execution of a distributed system of n independent nodes each of which executes actions on local objects at the request of a set of transaction instances may be modelled as the execution of a single node executing in some sequential order.

Proof: In the style of Lamport (see [Lamp 78]) local clocks are defined. Having these clocks a global schedule for the transaction instances is defined, which preserves the ordering within the transaction and system clocks. For the details, see [Tra et.al. 79].
QED

Having shown this one can apply the previous results to show that:
Theorem II.6.3

> If all transaction environments are well-formed and two-phase then the execution of the transaction by a distributed database system will be equivalent to some serial execution of the transactions by a centralized system.

Proof: See [Trai et.al.79].

II.6.1.2.2. Predicate locks

The preceding paragraph applies to any system which supports the concepts of transaction and shared entity. We will now concentrate on locking in a centralized database environment, which for definiteness is assumed to be relational. Later on the results will be extended to a distributed environment. The unit of locking is substantially different now: it is common that a transaction wants to lock the set of all entities satisfying a certain predicate. Updating a previously unrelated entity may add it to such a set. This is called the phantom record problem, which has to be solved to preserve consistency.
It is of course very inefficient to lock whole relations or domains whenever any member of the relation or domain is referenced, because this nearly rules out concurrency. This suggests that locks should apply to as small a unit as possible so that transactions do not lock information they do not need. Therefore the natural unit of locking is the field or tuple of a relation. But a tuple is not an entity in the sense of the preceding paragraph because it is referenced by value rather than by the address of the storage it occupies. Therefore, common LOCK requests are not suitable for locking tuples to be selected from a relation.
Example II.6.2

For the relation

ACCOUNTS (Location, Number, Balance) a baicliff addresses the request
SELECT FROM ACCOUNTS
WHERE Balance < -500.

Here the tuples are selected on account of their balance field, while a request of the form LOCK A on entity A is directed to a certain storage address.

This relation will be referred to in the forthcoming examples in this section.

End of Example

Example II.6.3

Even more elementary is the test for the existence of a tuple in ACCOUNTS with Balance < -500. If the tuple exists, it is to be LOCKed to insure that no other transaction will delete it before the first transaction terminates. If the tuple does not exist, we have to guarantee that no other transaction will create such a tuple before the first transaction terminates: in this case non-existent tuples, called phantoms ,must be locked.

End of Example

As proved in lemma II.6.1 a transaction must lock all tuples examined, both real and phantom. The arguments above show that it is natural to lock the set of tuples and phantoms satisfying a certain predicate. If P is a predicate on tuples of relation R then P defines the subset S where

t S iff P(t). Transactions will be allowed to lock any subset of a relation by specifying such a predicate, so that predicates become the unit of locking.

Example II.6.4

Locking the tuples of ACCOUNTS with Location = 'Washington' is achieved by
LOCK ACCOUNTS : Location = 'Washington'

End of Example

However, we cannot directly apply the formulation of locking and consistency of the previous section, because entities are not uniquely named anymore. Hence the results on scheduling and consistency must be extended so that thay apply to LOCKs on possibly overlapping sets.

It is a recursively unsolvable problem to decide whether two distinct predicates define overlapping sets, so predicate locks must be scheduled cleverer than that. In general the scheduler must compare a requested predicate lock against the outstanding predicate locks of other transactions on this relation. If two such predicates are mutually satisfiable then there is a conflict, and the request must wait or preempt.

For notational simplicity the lockable entity will be a tuple rather than a field for the time being. We proceed more formally.

Notation

An action on a single tuple t of relation R accessed in mode a, is denoted by (R, t, a). There are two modes:

1) a = READ: allows sharing
2) a = WRITE: requires exclusive lock on t.

Example II.6.5

An update of the Balance of account number 123 of Washington with Balance 50 by 30 involves two actions and two tuples, first
(ACCOUNTS, (Washington, 123, 50), WRITE) and then
(ACCOUNTS, (Washington, 123, 80), WRITE), because both tuples are written by the atomic update operation: the first is deleted, and the second is inserted.

End of Example

It is immediately clear that it is awkward to require an action to specify a tuple by providing all field values. Therefore the concept of action is generalized to that of access , which acts on all tuples satisfying a given predicate.

Example II.6.6

To access Account number 123, one issues the access
(ACCOUNTS, Number = 123, READ).

End of Example

Notation

An access on relation R is denoted by (R, P, a) where P is a predicate, and a again is the access mode. It is equivalent to the set of actions (R, t, a), where P(t) = TRUE and t underlying Cartesian product of R.

A predicate lock on relation R is denoted by (R, P, a), where P is a predicate and a is an access mode.

Definition II.6.15

An action (R, t, a) satisfies predicate lock (R', P', A') if
R = R'
P'(t) = TRUE
a = a' or a' = WRITE (WRITE access implies READ access).

An action (R, t, a) conflicts with predicate lock (R', P', a') if
R = R'
P'(t) = TRUE
a = WRITE and a' = READ.

End of Definition

These notions are defined analogously for accesses:

Definition II.6.16

An access A = (R, P, a) satisfies predicate lock L iff for each tuple t in the underlying Cartesian product C of R:
P(t) $\wedge$ (R, t, a) satisfies L.

An access A = (R, P, a) conflicts with predicate lock L if for some tuple t in C:
P(t) $\wedge$ (R, t, a) conflicts with L.

End of Definition

Definition II.6.17

Two predicate locks are said to conflict if there is some action which satisfies one of them and conflicts with the other.

End of Definition

Given these definitions the notions of the previous section can be generalized:

A transaction is a sequence of (transaction name, access) pairs.

A transaction is well-formed if each access it makes satisfies some predicate lock it holds through that step.

A transaction is two-phase if it does not request new predicate locks after releasing a predicate lock.

A schedule for a set of transactions is any merging of the transaction sequences.

An entity is a (tuple, relation) pair.

Let E denote the set of entities.

Definition II.6.18

The dependency set of a schedule S with set of transactions T is the set
(T1, e, T2) $\in$ T x E x T such that for some integers i,j:
i < j ,

S(i) = (T1, a1) and access A1 READs or WRITEs entity e,
S(j) = (T2, a2) and access A2 READs or WRITEs entity e,
not both A1 and A2 simply READ e and
for any k between i and j: if S(k) = (T3, A3) then A3 does not WRITE e.
End of Definition

We continue to generalize to field-level predicate locks. A particular field-level access to a relation READs, WRITEs or ignores each of the fields of the relation specified by the access predicate.

Definition II.6.19

Two field-level predicate locks are said to conflict if some action which satisfies both is a WRITE access for a field LOCKed in READ mode by the other.

End of Definition

Definition II.6.20

A field-level access satisfies predicate lock L if it only accesses tuples covered by L and
it only READs fields LOCKed in READ or WRITE mode by L and
it only WRITEs fields LOCKed in WRITE mode by L.

A field-level access conflicts with predicate lock L if the predicates are mutually satisfiable and the access
WRITEs a field of a tuple LOCKed in READ mode by L or
accesses tuples not covered by L. The latter guarantees that phantoms are prohibited.

End of Definition

Example II.6.7

Let A denote the access
(ACCOUNTS, Number = 123, {(Number, READ), (Balance, WRITE)}
and let L denote the field-level predicate lock
(ACCOUNTS, Number = 123, {(Number, WRITE), (Balance, WRITE)}.
Then A satisfies L.
Let L′ denote the access
(ACCOUNTS, Number = 123, {(Number, WRITE), (Balance, WRITE)}.
Then L is in conflict with L′: L has WRITE access for Number for which L′ has only READ access.
Further A is in conflict with L′: A accesses a tuple not covered by L′.

End of Example

After all this theory some practical remarks.
To implement arbitrary predicate locks, a table called SYSLOCK is associated with the database. This is a binary relation between transactions and predicate locks. The legal SYSLOCK scheduler enforces transactions to be two-phase and well-formed by consulting the SYSLOCK table: if a newly requested predicate lock does not conflict with any other predicate lock in the table it is implemented immediately. Otherwise the requestor must wait for the other locks to be released, or preempt the locks.
The question whether two predicate locks conflict or not is in general a recursively unsolvable problem. Therefore we need to find an interesting class of predicates for which it is easily decidable whether two predicates overlap.

Definition II.6.21

An atomic predicate is a predicate of the form

<field-name> $\left\{ \begin{matrix} = \\ \neq \\ < \\ \leq \\ > \\ \geq \end{matrix} \right.$ <constant>

e.g. Location = Washington.

A simple predicate is any Boolean combination of atomic predicates.
End of Definition

To decide whether two simple predicate locks
L = (R, P, a) and L′ = (R′, P′, a′) conflict is fairly straightforward:
If R <> R′ there is no conflict.
If neither a = WRITE nor a′ = WRITE there is no conflict.
Otherwise there is no conflict only if there is no tuple t such that
P ∧ P′(t) = TRUE.
Similarly for accesses.
This reduces the above test to the question of deciding whether a particular simple predicate is satisfiable. This can be done by bringing P P′ into disjunctive normal form. For the details, see [Esw et.al. 76].

In fact locking is a very dynamic form of authorization: the techniques of predicate locks and simple predicate locks apply to the problem of doing value-dependent authorization of access to the fields of database records.

II.6.1.3. Lock conversion

In the preceding two sections it is assumed that once a transaction acquires a particular kind of LOCK on a data item it is not allowed to convert this LOCK to another kind. Requiring that transactions be two-phase has the major drawback that it severely restricts the amount of concurrency allowed in the system. This flaw of two-phase lock (2PL) protocols can be resolved partially by providing more interpretation concerning the precise functions performed by a transaction: imagine the items of the database are organized in the form of a directed acyclic graph (DAG), with each vertex of the graph being an entity, and forcing the transactions to acquire LOCKs on entities in some restricted ways. The edges of the graph model logical or physical relationships. Adaption of this graph interpretation has led to the development of non-2PL protocols which allow more concurrency than 2PL protocols.
In [Moh et.al.83] a systematic study of allowing lock conversions in order to increase concurrency is performed. Serializability is again the correctness criterion for the protocols, which now allow transactions to convert a LOCK held on an item from one mode to another without unlocking it. This is done as follows:

- During the growing phase apart from the LOCK_S and LOCK_X actions a transaction can issue the UPGRADE action which upgrades a LOCK from LOCK_S to LOCK_X.
- During the shrinking phase apart from the UNLOCK action a transaction can perform a DOWNGRADE action which downgrades a LOCK from LOCK_X to LOCK_S.

All protocols now require that transactions be two-phase in an extended sense: once a transaction issues a shrinking instruction it may no longer issue a growing instruction on that data item.
Example II.6.8
If initially a LOCK_S is granted and just before a WRITE is done this LOCK is UPGRADEd then some transactions might have done a READ on the entity meanwhile.
End of Example

This and other examples show how concurrency can be enhanced, when lock conversion is allowed.
One of the most general non-2PL protocols is the Guard Locking Protocol (GLP),

of which several variations will be described.
Let us restrict to rooted DAGs, R being the root and V the set of vertices.

The Basic Guard Protocol (BGP)

Here a transaction may only issue a LOCK_X.

Definition II.6.22

A graph is guarded iff with each vertex $v \in V$ a non-empty set of pairs (called subguards) is associated,

$guard(v) = \{(A_1^v, B_1^v), \ldots (A_M^v, B_M^v)\}$

which satisfies:

1) $\emptyset <> B_i^v \subseteq A_i^v \subseteq V$
2) $\forall u \in A_i^v$ [u is a parent of v]
3) $A_i^v \cap B_j^v = \emptyset$ for every i and j.

End of Definition

Definition II.6.23

A subguard (A_i^v, B_i^v) is satisfied in mode m, $m \in \{S,X\}$ by transaction T iff T is currently holding a LOCK_m on all vertices in B_i^v and it had a LOCK_m on (and has possibly UNLOCKed) each vertex in $A_i^v - B_i^v$.

End of Definition

The rules of BGP for transaction T are:

1) Any vertex may be LOCKed first.
2) Subsequently, a vertex v can be LOCKed only if there is a subguard satisfied in mode X by T.
3) Vertices may be UNLOCKed any time.

Theorem II.6.4

BGP assures serializability.

Proof: See [Moh et.al. 83].

Theorem II.6.5

BGP assures deadlock-freedom.

Proof: See [Moh et.al. 83].

This is a great improvement on 2PL protocols.
BGP will be extended later to support also the S mode of locking.

Non-2PL protocols can be broadly divided into two types: heterogeneous and homogeneous.

The heterogeneous guard protocol (GLP') is characterized by the distinction being made between update and read-only transactions: update transactions may only acquire a LOCK_X, while read-only transactions may only acquire a LOCK_S. Further the update transactions must start by locking R. As a consequence the update transactions are serialized in the order in which they obtain a LOCK on R. Apart from that this protocol behaves the same as BGP. Therefore serializability is maintained.

Lemma II.6.4

Serializability cannot longer be guaranteed if update transactions may start by locking first another vertex than R.

Proof: Suppose the DAG is a tree: each transaction requesting a LOCK on a vertex v other than R must hold a LOCK on the parent of v. Suppose this relation among entities A and B holds:

[see Figure II.6.6]

Consider the following schedule S:

T1 LOCK_S R ; T1 LOCK_S A ; T1 UNLOCK R ; T2 LOCK_X B ;
T2 UNLOCK B ; T1 LOCK_S B ; T1 UNLOCK A ; T1 UNLOCK B . This schedule is not serializable since both (T1, A, T2) and (T2, B, T1) are in DEP(S).

QED

GLP' can be modified in a natural way so that update transactions are allowed

Figure II.6.6

to obtain a LOCK_S and to perform lock conversions. The modified protocol is called MGLP' and it proceeds as follows:

1) If there is a LOCK_X on the first entity then that entity must be the root of the DAG.
2) Subsequently, a vertex v can be UPGRADEd or DOWGRADEd only if there exists a subguard of v satisfied in X (respectively S or X) mode by T. Exception: a LOCK_S on R can be upgraded at any time.
3) A read-only transaction can obtain its first lock on any vertex. Vertices can be UNLOCKed or DOWNGRADEd any time.

Theorem II.6.6
MGLP' assures serializability.
Proof: See [Moh et.al. 83].

The homogeneous guard protocol (also called the pitfall protocol) does not distinguish between read-only and update transactions. Instead, any transaction may acquire both a LOCK_S and a LOCK_X on different items, and start locking any vertex first. But this protocol requires the transactions to be two-phase during certain segments of their locking actions.
We first need a few definitions.
Notation
Let L(Ti) be the set of all entities LOCKed by Ti, LX(Ti) be the set of entities LOCKed in mode X by Ti and LS(Ti) be the set of entities LOCKed in mode S by Ti. A LOCK is held on each member of these sets by Ti some time during its execution.
Definition II.6.24
Consider the subgraph of the tree spanned by the set LS(Ti). A pitfall of Ti is a set of the form $A \cup \{v \in LX(Ti) \mid v \text{ is a neighbour of some } w \in A\}$ where A is a connected component of the tree.
End of Definition

Definition II.6.25
A transaction Ti is said to be two-phase on a set of entities iff T_i^K, the set of actions of Ti referring to only the vertices in K, is two-phase: no LOCK on an entity is acquired after an entity has been UNLOCKed.
End of Definition

The rules for the pitfall protocol for transaction T are:

1) The first LOCK can be acquired on any vertex.
2) Subsequently, a vertex v can be LOCKed only if there is a subguard satisfied by T (each vertex in the subguard may have been LOCKed in any mode).
3) The transaction must be two-phase on each pitfall.

Theorem II.6.7
The pitfall protocol assures serializability.
Proof: See [Moh et.al. 83].

The pitfall protocol can be extended too by permitting lock conversion.
Notation Let LSt(T) be the set of all entities on which T at step t either had

a LOCK_S or had a LOCK_S before they were UNLOCKed. The set LXt(T) is similarly defined.

Definition II.6.26

An M-pitfall of T at time t is the union of the set of entities in a maximal connected component of LSt(T) and the set of its neighbours in LXt(T).

End of Definition

The rules for the extended pitfall protocol are:

1) Any vertex can be LOCKed first.
2) Subsequently, a vertex v can be LOCKed only if there is a subguard satisfied in either X or S mode by T.
3) T must be generalized two-phase on its M-pitfalls: T can LOCK or UPGRADE a LOCK on an entity e in some M-pitfall P only if it did not UNLOCK or DOWNGRADE a LOCK on any entity in P.

Theorem II.6.8

The extended pitfall protocol assures serializability.

Proof: See [Moh et.al. 83].

By theorem II.6.2 BGP, GLP', MGLP', the pitfall protocol and the extended pitfall protocol generalise to distributed systems.

II.6.1.4. Optimistic methods for concurrency control

The locking approach of the preceding three sections has the following inherent disadvantages:

a) extra overhead, even for read-only transactions which cannot possibly affect data consistency
b) to allow a transaction to abort, LOCKs cannot be released until the end of the transaction
c) if large parts of the database are on secondary storage, concurrency is lowered whenever an often accessed node has to remain LOCKed while waiting for a secondary memory access
d) locking is often not necessary and is only a worst case need. In general, this holds whenever there are many more nodes than the number involved in the transaction and the probability of modifying an often used node is small.
e) a general purpose locking protocol that always provides high concurrency can cause the problem of deadlock.

For these reasons the idea of eliminating locking rises.
In [KuRo 81] concurrency protocols are proposed that do not use locking. These protocols are optimistic in the sense that they rely for efficiency on the hope that conflicts between transactions will not occur.
The idea behind the optimistic approach may be formulated as follows:

1) Since a READ action can never cause inconsistency READs are completely unrestricted.
2) WRITE actions are severely restricted. It is required that any transaction consists of two or three phases: a READ-phase, a VALIDATE-phase and a possible WRITE-phase. During the READ-phase all WRITE actions take place on local copies of the entities to be modified. After the VALIDATE-phase, only if it turns out that the performed updates will not cause inconsistency, the local copies are made global in the WRITE-phase. The correctness criteria used in the VALIDATE-phase are again based on serializability.

The READ and WRITE-phases

Objects are manipulated by the procedures CREATE, DELETE, READ and WRITE. Transactions are written by the user as if these procedures were used directly. However, the system requires that transactions use the syntactically identical procedures TCREATE, TDELETE, TREAD, and TWRITE. For each transaction the concurrency control maintains sets of objects accessed by the transaction. A transaction starts with executing TBEGIN in which these sets are initialized.
The body of a user-written transaction is in fact the READ-phase. When a transaction completes it calls the procedure TEND, which performs the VALIDATE and WRITE-phase.

The VALIDATE-phase

By definition II.6.10 a permutation Π 17must be found. This is done by explicitly assigning each transaction Ti a unique transaction number t(i) during execution. The meaning of these numbers in the VALIDATE-phase is: whenever t(i) < t(j) there is an equivalent serial schedule in which Ti precedes Tj. We have the following validation condition:
For each transaction Tj with transaction number t(j) and for all Ti with t(i) < t(j) one of the following statements must hold [Pap 79]:

a) Ti completes before Tj starts.
b) Ti's WRITE-phase does not affect Tj's READ-phase, but Ti starts its WRITE-phase before Tj finishes its WRITE-phase.
c) During the WRITE-phase of Ti none object is written which is read or written by Tj, and Ti completes its READ-phase before Tj completes its READ-phase.

For the assignment of transaction numbers a global integer counter tnc (transaction number counter) is maintained. When such a number is needed tnc is incremented and the resulting value is returned. The assignment must occur somewhere before the VALIDATE-phase. In the optimistic approach we like transactions to be validated as soon as possible. Therefore the assignment is at the end of the READ-phase.
A further optimization is that transaction numbers are assigned only if the VALIDATE-phase succeeds.

There are two families of optimistic concurrency control protocols.

Protocols with a serial VALIDATE-phase

They implement validation conditions a) and b). Since condition c) is not used the last part of condition 2) implies that WRITE-phases must be serial. This can be achieved by placing the assignment of a transaction number, the VALIDATE-phase and subsequent WRITE-phase all in a critical section. See [KuRo 81] for the details.

Protocols with a parallel VALIDATE-phase

These protocols use all three of the validation conditions, and thus allow greater concurrency. For condition c) a set ACTIVE of transaction-id's is maintained of those transactions that have completed their READ-phase but not yet completed their WRITE-phase. For the details see [KuRo 81].

By theorem II.6.2 these optimistic protocols generalise to distributed systems.

II.6.2. Replicated data and multiple copy update

There are several arguments for (partially or fully) redundant data:
- if a file is stored at many sites then the availability of the data for a

read request is higher than if the file were stored only once
- locality of reference is enhanced which is especially advantageous for frequently accessed data
- the opportunity to distribute the workload on a frequently accessed file
- greater reliability through backup copies in case of storage media failures.

Assume that for these reasons we use replicas. Then the notion of consistency incorporates two aspects:
1) internal consistency of redundant copies
2) mutual consistency of each copy.

Methods to preserve the internal consistency of a (copy of a) (partition of a) distributed database were described in the chapter on consistency preserving concurrency protocols.
As for the second aspect, copies of a database are said to be mutually consistent if they are identical. Since this is hard to achieve for every instant of time, this constraint is usually relaxed: multiple copies must converge to the same final state once all update actions are finished. Therefore, having replicas causes WRITE-requests to be more expensive. From the above it follows that the multiple copy update strategy is crucial for this aim: it must guarantee that an update is processed at each site containing a replica. This must be distinguished from a commit, where each site approves of the update actions of a transaction it has performed.

A description of various multiple copy update strategies follows. A general goal of these strategies is to hide the duplicates from the user i.e. the user's viewpoint is as if there is a single copy image of each data item. This is called distributive transparency for the user.

II.6.2.1. Multiple copy update strategies

II.6.2.1.1. Unanimous agreement update strategy

This is the easiest update strategy: updates are propagated to all replicas immediately . Unanimous acceptance of the proposed update by all sites having replicas is necessary in order to make a modification and therefore all of those sites must be available. Hence when the number of copies increases, the probability that an update succeeds decreases. This is a bad property.
All following strategies do not exhibit this flaw.

II.6.2.1.2. Single Primary Update Strategy

Here one replica is designated as PRIMARY, the remaining replicas as SECONDARY. Update requests are issued to the primary replica which serves to serialize updates and thus preserve data consistency.
After having performed the update, the PRIMARY will broadcast it to each SECONDARY. This causes a delay which may be perceptible to a user: if a user issues an update followed by a read and the update is at a remote site while the read is local there is a chance he will miss the update. This is called the problem of D elayed U pdate N otification (D.U.N.).

II.6.2.1.3. Moving Primary Update Stategy

Again there is one designated PRIMARY site, the others being SECONDARY sites. Update requests are made to the PRIMARY or any SECONDARY. In general updaters are unaware which site is functioning as the PRIMARY at a particular time.

When an update request is received, and the receiving site is the PRIMARY, then the update is performed and the PRIMARY sends a cooperation request to a SECONDARY, informing it of the update. The secondary performs the update, sends an ACK to the PRIMARY and to the original requestor, and finally passes the request to the next SECONDARY. Once the PRIMARY has received the ACK from the secondary it is certain that the update will not be lost unless both the PRIMARY and the SECONDARY concerned fail. This is called a 2-host resilient scheme. In general, a m-host resilient scheme can tolerate m failures.
If the request is received by a SECONDARY, it forwards the request to the PRIMARY which proceeds as above.
If the PRIMARY fails, it will be discovered by a SECONDARY when it forwards its next request or by the requesting site. Upon discovery the secondaries begin to recoverably elect a new PRIMARY from amongst themselves. Any existing election algorithm is suitable for this. In a 2-host resilient scheme, each SECONDARY must participate in the election, while in a m-host resilient scheme any m SECONDARYs can be left out of it.
The D.U.N. problem appearing in the preceding strategy also occurs for this scheme.

II.6.2.1.4. Majority Vote Update Strategy

This proposal involves designating all sites with a replica as peers. An update succeeds if it is accepted by a majority of peers. As soon as a site or link failure occurs this majority changes. Each majority which agreed on a set of updates is uniquely identified by a so called version number as follows: after a change of majority the new majority will have a higher version number than the old one. Every two majorities have at least one member in common who will insist that the other members of the newly formed majority will pick the latest version of the data before any updates are accepted.
Thus, successive updates may be accepted by different majorities, but only one majority at a time will be processing updates and these updates will always be applied to the latest version of the data.
When a site has an update request, it must broadcast update request messages to all members of the last majority in which it participated. There are several possibilities now:

- that majority is still ruling and the request will succeed
- that majority is no longer viable and the requesting site is the first to detect that. It will attempt to form a new majority.
- meanwhile a new ruling majority has been formed. The requesting site requests to join that majority.

When a node is only allowed to participate in an update iff it has the latest version of the replicated data, the D.U.N. problem of the preceding strategies is ruled out.

II.6.2.1.5. Majority Read Update Strategy

So far the update work was forced to be done at update time. This scheme places the work at the time of a read request. It performs update requests on a replicated file at the local site of request. The update is then broadcast to whatever sites are on-line. The update request succeeds if it accepted by a majority of peers. To each update a version number is assigned which is monotonically increasing. Each peer accepts updates only in order of this number and answers the read request along with the version number of the last update it has seen.
This approach is only feasible when messages are fast and cheap and then still the overall system is only available to update requests as long as there is a majority of peers which are on-line.

II.6.2.2. Remarks on data consistency

In the chapter on consistency preserving concurrency protocols it was shown how data in a distributed database can achieve the appearance that all data is stored as a single copy at a single site. For replicated data two additional properties are required:

1) updates must be broadcast to all replicas before the transaction commits
2) if updates are not immediately broadcast and performed by all replicas then all accesses after an update must be to an updated copy to avoid the D.U.N. problem.

In several multiple copy update strategies however, property 1 is relaxed: some peers may not receive updates at the end of an updating transaction. This justifies the notion of currency for a read request. A currency read request (a wish to read the most current value) is a more expensive operation than an ordinary read request. The value of such an ordinary read request is the current value as of the time of the last update request which was performed on the replica satisfying the read request. Consequently, if the site is PRIMARY or in the ruling majority, then an ordinary and currency read request yield the same value.
On the other hand, a site, which has a replica but is SECONDARY or has not participated in the latest update majority, may continue to read the older data so long it does not also access newer data or any data derived from newer data.
So we have to prohibit that a read at a 'new' site is followed by a read at an 'old' site, which would also occur in case of a D.U.N. The data seen in these situations is inconsistent which is an obstruction of the aim at distributive transparency for the user.
To avoid that old and new sites interleave in the same transaction, the version number and majority list for each replicated file are exchanged and compared whenever a transaction makes a request to another site. In this way transactions keep track of the relative newness of the data they have accessed so far. A transaction refuses to run at sites known to contain older data, and must be aborted if it encounters newer data because this causes inconsistency with the data previously seen.

II.7. Some special purpose developments in transaction management

The particular recovery manager of System R is described in [Gray 79].

In [Lis 84] an overview is presented of an integrated programming language and system, called Argus, specially designed to support distributed programs (programs running on a network of computers). The various functions of transaction management are all implemented.

In [Ell 77] a formal validation technique is developed and applied to the duplicate database problem.

II.8. Concurrency in heterogeneous DBMSs

In [GlPo 85] special attention is paid to the heterogeneity aspect of distributed DBMSs.
Current distributed heterogeneous DBMSs address the issue of concurrency control in two ways:

a) Some rely solely on unmodified local concurrency control mechanisms. This helps in preserving site autonomy, but limits functionality to various degrees (e.g. allow only unsynchronized retrievals, preclude distributed

updates, perform local updates off-line etc.).

b) Others maintain full functionality but require the (re)design of the local mechanisms to enforce homogeneous concurrency control accross all heterogeneous DBMSs, thereby giving up site autonomy.

A set of conditions is presented necessary to establish a global concurrency control mechanism based on the concatenation of local ones. This mechanism allows synchronized retrievals and distributed updates and still maintains a high degree of site autonomy.
Let DTM denote the distributed transaction manager and LTM denote a local transaction manager. The conditions are:

1) The concurrency control mechanisms of all LTMs provide local serializability.
2) The LTMs preserve the relative order of execution of actions determined by the DTM.
3) Each distributed transaction may perform only one action at a given site.
4) The DTM must be able to identify all objects referenced by all actions.
5) The DTM must be able to detect and recover from global deadlock.

Definition II.8.27

A TWF (transaction-wait-for)-graph is a graph in which each vertex represents a transaction, and a directed edge (Ti, Tj) indicates that Tj has to wait for Ti.

End of Definition

Given this notion the last condition splits into:

a) All concurrency control mechanisms of LTMs must allow the derivation of local TWF-graphs.

b) All local concurrency control decisions which cause local transaction waits or aborts, and the status of local entities (e.g. LOCKed or not) must be visible to the global deadlock detector of the DTM.

The ability to cope with the other aspects of transaction management will be decisive for a possible introduction of distributed heterogeneous DBMSs.

Conclusions

The security of distributed DBMSs involves two aspects:

a) a practical one so as to give only properly authorized users access rights.
b) a theoretical one so as to give concurrent (legal) users the ability to retrieve or enter data even in the presence of failures.

I have described many protocols which provide this security. As computers in general and databases in particular are used more and more in our society, their importance will only increase in future.

References

After each reference I have indicated the section(s) where it has been cited.

[AdAz 85] Adiba, M., and F.Azrou, An authorization mechanism for a generalized DBMS, in 'Distributed Data Sharing Systems III', North Holland, 1985, pp.137-153. Ch. I.4.

[Cha et.al. 77] Chamberlain, D.D., Gray, J.N., Griffiths, P.P., Traiger, I.L., Wade, B.W., Database system authorization, IBM Research Report RJ 2041, IBM Research Laboratories, San Jose, California, july 1977. Ch. I.3.

[Date 81] Date, C.J., An introduction to Database Systems, 3rd edition, Addison Wesley Publ.Comp., 1981. Introduction, Ch. I.5.

[DeMi 82] DeMillo, R.A., N.A. Lynch, and M.J. Meritt, Cryptographic Protocols, Proc. 14th Ann. ACM Symp. Theory of computing, 1982, pp. 383-400. Ch. II.6.

[DoSt 82] Dolev, D., and H.R. Strong, Authenticated Algorithms for Byzantine Agreement, IBM Research Report RJ 3416, IBM Research Laboratories, San Jose, California , march 1982. Ch. II.6.

[Ell 77] Ellis, C.A., Consistency and Correctness of Duplicate Database Systems, Operating Systems Review vol.11,5, november 1977. Ch. II.7.

[Esw et al. 76] Eswaran, K.P., J.N. Gray, R.A. Lorie, and I.L. Traiger, The notions of consistency and predicate locks in a database system, CACM vol.19,11, november 1976. Ch. II.6.

[GlPo 85] Gligor, V.D., and R. Popescu-Zeletin, Concurrency control-issues in distributed heterogeneous database management systems, in the same book as [AdAz 85], pp.43-57. Ch. II.8.

[Gray 78] Gray, J.N., Notes on Database Operating Systems, An advanced course, Bayer, Graham and Seegmuller, (eds.), Lecture Notes on Computer Science vol.60, Springer Verlag, 1978, pp.393-482. Ch. II.5, II.6.

[Gray et.al. 79] Gray, J.N., P. McJones, M. Blasgen, R. Lorie, T. Price, F. Putzolu, and I. Traiger , The recovery manager of a data management system, IBM Research Report RJ 2623, IBM Research Laboratories, San Jose, California, august 1979. Ch. II.7.

[GrWa 76] Griffiths, P.P., and B.W. Wade, An authorization mechanism for a relational database system, ACM Trans. on Database Sys., vol.1,no.3, sept 1976. Ch. I.3.

[Hol 81] Holler, E., Multiple copy update, Distributed Systems: Architecture and Implementation, An advanced course, Paul, Graham and Siegert, (eds.), Lecture Notes on Computer Science vol.105, Springer Verlag, 1981, pp.284-308. Ch. II.6.

[KuRo 81] Kung, H.T., and J.T. Robinson, On optimistic methods for concurrency control, ACM Trans. on Database Sys., vol.6,no.2, june 1981. Ch. II.6.

[Lamp 78] Lamport, L., Time, Clocks and the Ordering of events in a distributed system, C.ACM, vol.21,no.7, 1978, pp.558-565. Ch. II.6.

[Lam 81] Lampson, B.W., Atomic Transactions, in the same book as [Hol 81], pp.246-265. Ch. II.6.

[LeLa 81] Le Lann, G., Synchronization , in the same book as [Hol 81], pp.266-283. Ch. II.6.

[Lin et.al. 79] Lindsay, B.G., P.G. Selinger, C. Galtieri, J.N. Gray, R.A. Lorie, T.G. Price, F. Putzolu, I.L. Traiger, B.W. Wade, Notes on distributed databases, IBM Research Report RJ 2571, IBM Research Laboratories, San Jose, California, july 1979. Ch. II.2, II.3, II.4, II.5.

[Lisk 84] Liskov, B., The Argus Language and System, An advanced course,

M.Paul and H.J. Siegert, (eds.), Lecture Notes on Computer Science vol.190, Springer Verlag, 1984. Ch. II.7.

[Moh et.al. 83]
Mohan, C., D. Fussell, Z. Kedem, and A. Silberschatz, Lock conversion in non-two-phase locking protocols, IBM Research Report RJ 3947 , IBM Research Laboratories, San Jose, California, july 1983. Ch. II.6.

[MoLi 83] Mohan, C., and B. Lindsay, Efficient commit protocols for the tree of processes model of distributed transactions, IBM Research Report RJ 3881, IBM Research Laboratories, San Jose, California, june 1983. Ch. II.5.

[Pap 79] Papadimitriou, C.H., The serializability of concurrent database updates, J.ACM, vol.26,no.4, oct.1979, pp.631-653. Ch. II.6.

[PlvL 85a]
Pluimakers, G.M.J., and J. van Leeuwen, Authentication, Technical Report RUU-CS-85-9a, Dept. of Computer Science, University of Utrecht, 1985. Ch. I.1, II.3.

[PlvL 85b]
Pluimakers, G.M.J., and J. van Leeuwen, Authentication, a concise survey, to appear in 'Computers and Security'. Ch. I.1, II.3.

[Tra et.al. 79]
Traiger, I.L., J.N. Gray, C.A. Galtieri and B.G. Lindsay, Transactions and Consistency in Distributed Database Systems, IBM Research Report RJ 2555, IBM Research Laboratories, San Jose, California, june 1979. Ch. II.6.