A COMPARATIVE STUDY OF TRANSACTION MANAGEMENT SERVICES IN MULTIDATABASE HETEROGENEOUS SYSTEMS

by

KAREN VERA RENAUD

submitted in part fulfilment of the requirements
for the degree of

MASTER OF SCIENCE

in the subject

COMPUTER SCIENCE

at the

UNIVERSITY OF SOUTH AFRICA

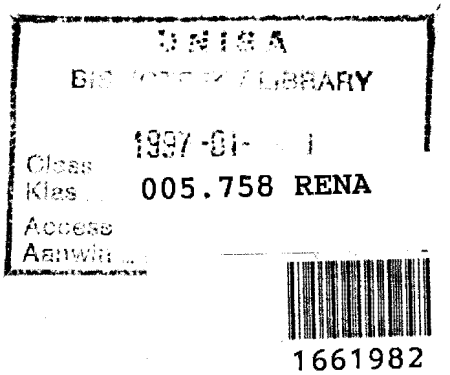
SUPERVISOR: MRS P KOTZÉ

APRIL 1996

Abstract

Multidatabases are being actively researched as a relatively new area in which many aspects are not yet fully understood. This area of transaction management in multidatabase systems still has many unresolved problems. The problem areas which this dissertation addresses are classification of multidatabase systems, global concurrency control, correctness criterion in a multidatabase environment, global deadlock detection, atomic commitment and crash recovery. A core group of research addressing these problems was identified and studied. The dissertation contributes to the multidatabase transaction management topic by introducing an alternative classification method for such multiple database systems; assessing existing research into transaction management schemes and based on this assessment, proposes a transaction processing model founded on the optimal properties of transaction management identified during the course of this research.

Key terms:
Multidatabase; Recovery; Reliability; Autonomy; Heterogeneity; Distribution; Concurrency control; Atomic Transaction; Transaction; Serializability; Global serialization; Global transaction atomicity; Global deadlock; Two-phase commit protocol; Compensation; Recoverability; Strictness.



To my delightful sons,

Gareth Lloyd, Ashley Norman and Keagan Philip.

Acknowledgements

- All glory be to my Lord and Saviour Jesus Christ from whom all good gifts come and without whom this work would never have been completed.
- Special thanks to my husband, Leon, for his support, encouragement, patience, tolerance and above all, his love and understanding.
- I am especially grateful to my supervisor, Paula Kotzé for her superb academic supervision, support, motivation, many hours of hard work, and for her unflagging patience.
- Heartfelt thanks to my Aunt, Felicity Howard, for proofreading this dissertation.

Contents

List of Figures

List of Definitions

List of Examples

List of Synopses

Chapter 1

Introduction

Research and development during the last decade have made great strides towards making distributed databases a commercial reality. A number of products are already readily available on the market and more are being introduced. Stonebraker *et al* [Sto94] claim that in the next 10 years there will be such a significant move toward distributed data managers that centralized data managers will become an "antique curiosity".

Distributed databases are an ideal means of sharing data and resources without affecting the autonomy of the database systems comprising the distributed database system. When distributed database systems are built in a bottom-up fashion from pre-existing database systems, we have a multidatabase system which can be loosely defined as an interconnection of autonomous database systems [Bar90]. This dissertation addresses this special type of distributed database system. A more precise definition of the multidatabase concept is given in Chapter 2.

1.1 Why a Study on Multidatabases?

More and more applications today require access to data residing in multiple, geographically distributed information stores. As a result, the integration of these stores has assumed some importance in recent years and the rapid development in the networking technology has made this integration tenable.

There is now effectively one world-wide telephone system and one world-wide computer network. Some people talk about a world-wide file system where data will be available to everyone anywhere. Likewise, we can contemplate a world-wide database system from which users could obtain data on any topic covered and where data is made available for public use. This type of application could be quite far away but it is necessary to start developing the technology for it now.

In many instances, the information sources are pre-existing database management systems operating in heterogeneous hardware and software environments, and following different protocols for concurrency control and recovery. Multiple database systems are an

important research area and the research into this area is expected to gain momentum with the advent of scientific and CAD/CAM(Computer Aided Design/Computer Aided Manufacturing) database applications. In a recent report of the NSF Workshop in Future Directions in database management system (DBMS) Research [Lag90], the area of heterogeneous, distributed database was identified as one of the two most important research areas in the 90's. [Sil83, Ras93b]

There are a number of applications that are now becoming feasible and that will help drive the technology needed for worldwide interconnection of information [Sil83]:

- Collaborative efforts are under way in many physical science disciplines, entailing multiproject databases. The project has a database composed of portions of research assembled by independent researchers. The human genome[1] project is one example of this phenomenon.

- A typical defence contractor has a collection of subcontractors assisting with portions of the contractor project. The contractor wants a single project database that spans the portions of the project database administered by the contractor and each subcontractor.

- An automobile company wishes to allow suppliers access to new car designs under consideration. In this way, suppliers can give feedback on the cost of components. This feedback will allow the most cost-effective design and manufacturing method for the car. This requires a database that spans multiple organizations.

While the need for integrating pre-existing systems is accepted and well understood, the difficulties inherent in this approach are still being investigated. The integration of multiple pre-existing databases is not a trivial task as one does not want to integrate them and rewrite all applications, but rather simply access the data without interfering with the autonomy, ownership, security considerations and unique features of each particular DBMS. This dissertation addresses the concurrent transaction management aspects of these multidatabase systems.

1.2 Database Background

Databases were originally introduced in order to collect all the data in an organization into a sort of reservoir of data so that all data access could be done via a kit of database access tools, such as data description languages, data manipulation languages, access mechanisms, constraint checkers and high level languages. This kit of tools which controls all access to the database is referred to as the database management system (DBMS). The physical database together with the DBMS is called the database system.

[1] the genetic material of an organism

However, after providing this centralized database service to users, it was found, over time, that the situation was far from satisfactory. Some users did not want to lose control of their data, and dynamic tailoring of data structures to suit many different users became increasingly difficult to provide. Because of these factors, databases gradually became more decentralized with each department once again having their own database. This resulted in difficulties with communicating data between users, which in turn gave rise to the need for a more formal approach to the decentralization of databases and database functions while maintaining an integration of resources and perhaps a certain measure of centralized control [Bel92].

This decentralization trend has led to the development of *multiple database systems* (MDBS). A multiple database system typically consists of a software layer (ie. the DBMS) built on top of a set of multiple pre-existing database systems. Each individual database system of the set of multiple databases can be referred to as a *component* database system.

Transactions[2] can be submitted to a DBMS of a component database system of the multiple database system either directly, or via the software layer above the multiple database systems, or both, depending on the implementation alternative chosen.

The transactions submitted directly to a component database system are referred to as *local transactions* and the users of the component database systems are called *local users.* The transactions submitted to the multiple database system software layer are referred to as *global transactions* and the users who submit them are called *global users.* The databases in the component database systems are also sometimes referred to as local databases. The entire multiple database system is referred to in the literature as a multidatabase.

The multidatabase concept can be best illustrated by means of an example. In Figure 1.1 we illustrate a typical multiple database system. This multiple database system represents the computer setup in a company which has decided to purchase three pharmacies, one each in Pretoria, Cape Town and Port Elizabeth. The new company needs to access the data records of three pharmacies with three different database systems:

- Tonic Pharmacy, situated in Pretoria, which currently uses a relational database system,
- Medilots Pharmacy, situated in Cape Town, which currently uses a network database system and
- Harbour Pharmacy, situated in Port Elizabeth, currently using an object-oriented database system.

Each individual pharmacy database system must still continue to function as it always has but in addition we need to do global queries which combine data from all three sites as well as update and retrieval type transactions on the data in all the databases.

[2]Operations on data items in the database

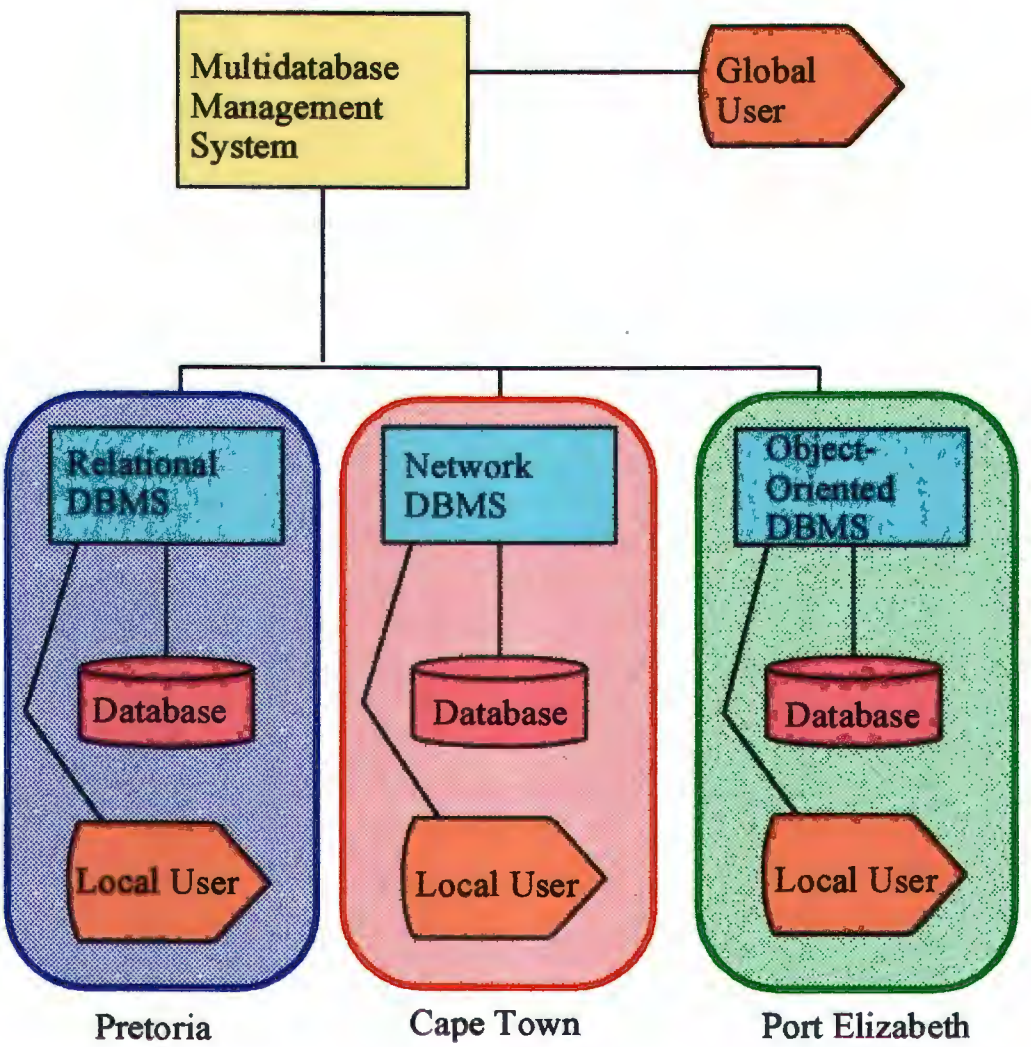


Figure 1.1: Overall architecture of the example multidatabase system

For instance, the new company needs to access all three databases to check sales; to do stock control; to keep tabs on the Schedule 6 and 7 medicines and to generate monthly accounts. The individual database systems must still be able to execute local transactions to register sales, issue prescriptions and enter purchases on customer accounts. Because the database systems at each site are autonomous and pre-existing, we choose not to standardize them to all use the same DBMS and underlying database structure.

Each pharmacy database system is independent, and continues to function as it always has, with local users submitting transactions, and the local database administrator (DBA) admitting users and doing all the things for the database that he/she regularly does.

1.3 Aim, Structure and Achievements of the dissertation

The research into multidatabases has been concentrated on schema integration, semantic heterogeneity, transaction management and query optimization. In this dissertation we look at the topic of transaction management in heterogeneous distributed database systems which is a difficult issue. The main problem is that the various independent database systems comprising a multidatabase system will probably have different DBMSs and therefore will use different concurrency control, global commit and crash recovery schemes. A fair amount of work has been done into the global concurrency control problem. Very little has been done in the field of reliability and recovery of multidatabases and therefore this dissertation also addresses these aspects of transaction management in multidatabase systems.

In this dissertation an attempt has been made to define a model for transaction management in a heterogeneous distributed database system based on an in-depth study of past research in this area.

Chapter 2 defines the multidatabase concept and Chapter 3 introduces the basic concurrency control and recovery concepts in centralized database systems.

A core group of existing transaction management schemes which propose different mechanisms for transaction management in multidatabase systems has been selected and has been introduced in Chapter 4.

Most of the research efforts which form the core group have been done by what is considered to be a leading contributor in this area (Barker & Özsu; Breitbart, Olson, Thompson, Silberschatz, Georgakopoulos, Rusinkiewicz, Litwin & Garcia-Molina; Elmagarmid, Helal, Leu, Du, Litwin & Rusinkiewicz; Chen, Bukhres & Sharif-Askary; Garcia-Molina & Salem; Pu). Some relative newcomers have been included in the core group because they provide a new approach to the transaction management problem (Kang & Keefe and Yoo & Kim). This is by no means a complete list but serves to give an indication of related work done in the area of transaction management in multidatabases.

The respective concurrency control schemes of each of the core group's transaction management schemes are also discussed in this chapter. The field of global concurrency

control in multidatabase systems has been given a great deal of attention and for the sake of completeness, I have given brief synopses of various other concurrency control schemes that have been proposed in the literature.

The global deadlock issue has been briefly touched upon but since it is not included in the central theme of this dissertation, it is not discussed exhaustively.

Chapter 5 discusses global commit protocols and Chapter 6 discusses the crash recovery protocols used in the core group's transaction management schemes, when details of the global commit and recovery protocol are available in the literature. These fields have not enjoyed much attention from researchers into multidatabase concepts and the literature is quite sparse. In Chapter 7, the various transaction management schemes are summarized and finally evaluated and a model for transaction management in multidatabase systems is proposed.

1.3.1 A summary of problems addressed

This dissertation studies the following key problems in multidatabase systems:

- Classification of multidatabase systems — There are presently three distinct classification taxonomies for multidatabase systems. One which classifies multidatabases according to architectural differences [Bel92], another which classifies them according to degree of autonomy, heterogeneity and distribution [Özs90] and yet another which classifies them according to how tightly the participating local databases are coupled [Bri92]. This can cause some confusion and this problem has been addressed in Chapter 2 by introducing an alternative classification combining three taxonomy schemes.

- Transaction management — Various transaction management approaches have been proposed. The approach which has gained favour is the client-server approach started by Breitbart *et al* [Bre95]. There have been articles recently which advocate a totally decentralized global transaction manager which is located at each participant database system [Hwa94, Bat92, Kan93, Ye94].

- Global concurrency control — At first global serializability was the accepted method of ensuring correctness of concurrently executing global multidatabase transactions. Lately, however, various schemes have been proposed which do not maintain global serializability but which define other correctness criteria because global serializability is often seen as too restrictive. The various proposals are considered in Chapter 4.

- Global deadlock detection — This field has not received much attention. The latest work in the field is summarized in Chapter 4. The latest method, proposed by [Nam93] proposes using a graph structure in order to detect cycles which may exist and if they do exist, to resolve them.

- Global commitment — Many multidatabases use a variation of the two-phase commit protocol (discussed in Appendix C) and assume that the local database systems will support a prepare-to-commit state. In some systems, all transactions have to declare their data needs in advance and the commit process is then handled by the global transaction manager by controlling submission of operations. The latest trend which seems to be gaining favour is the one used by the decentralized and client-server type systems which handle global commitment by allowing their server or local agent to provide a prepare-to-commit state. The server or agent acts as a go-between between the global transaction manager and the local DBMS. This issue has been addressed in Chapter 5.

- Global crash recovery — Crash recovery is a question that has not had much attention from researchers. There are three approaches, redo, retry and compensate. A fair amount of research has been done into compensation but compensation is not viable in every type of system. Some recovery protocols require an exclusive access period when a site comes up after a crash [Geo91a]. The decentralized and client-server architectures will allow the local agent to control crash recovery [Yoo95]. This issue is discussed in Chapter 6.

- Recoverability of global transactions — Determining conditions under which multidatabase consistency can be ensured is not a trivial task. This concept goes hand in hand with the correctness criterion used by the multidatabase system. This issue is also addressed in Chapter 6.

- Correctness criterion — Various correctness criteria for multidatabase systems have been proposed by different researchers. The m-serializability correctness criteria seems to be seen by researchers as a reasonable alternative to the rather restrictive global serializability. This issue is touched upon in Chapter 4 and then again in Chapter 6.

1.3.2 A summary of achievements

The dissertation achieved the following:

- A multiple database system classification taxonomy was derived from three existing classification methods and is presented in Chapter 2.

- The multidatabase concept is formally defined in Chapter 2.

- A formal transaction model is presented in Chapter 3 and extended in Chapters 4 and 6 to include multidatabase concepts.

- A core group of divergent transaction management schemes has been chosen for this study and their various distinguishing features have been summarized in Chapter 4.

- A brief synopsis is given of recent research into various global concurrency control protocols and correctness in Chapter 4.
- A summary is given of recent research into global deadlock detection in Chapter 4.
- The need for global commit protocols in multidatabase systems is outlined and the various global commit protocols used by the core group are described and analyzed in Chapter 5.
- Crash recovery in the core group is elaborated upon in Chapter 6 where the concept of recoverability in multidatabase systems is also addressed.
- In the penultimate chapter, the eight transaction management schemes comprising the core group are evaluated and compared with one another.
- A formal transaction model — A transaction model was introduced by Bernstein [Ber87]. This transaction processing model has been extended by incorporating the work of Tang [Tan93], Mehrotra *et al* [Meh92c] and Barker [Bar90].
- Finally, a recommendation is made as to the transaction management scheme which seems to best satisfy the autonomy requirements in multidatabase systems. All the transaction management, global commitment and crash recovery schemes are evaluated and one scheme has been advanced as being the best at the moment. Reasons are given for the choice.

Chapter 2

The Multidatabase Concept

This chapter will introduce the multidatabase concept. The three dimensions defining multidatabases are discussed and a quantification method for the autonomy dimension is introduced. A classification for multiple database systems is introduced and the multidatabase concept is formally defined and elaborated upon.

2.1 Characteristics of Multiple Database Systems

There are three features which characterize multiple database systems: *distribution, heterogeneity* and *autonomy* [Özs90].

2.1.1 Distribution

The distribution characteristic deals with the location of data. Two cases can be identified [Özs90]. The data is either physically distributed over multiple sites or stored at one site:

- *Physically distributed* — this means that the software controlling the access to the multiple databases must utilize a network to communicate with the individual database system's DBMS.
- *Stored at one site* — the multiple database system software level does not need to use a communication medium to communicate with each component of the multiple database system but simply performs a logical integration of all the component database systems at one site. This type of multiple database system does not have to deal with problems involving failure of communication mediums and delay in responses inherent in a geographically distributed database system.

2.1.2 Heterogeneity

Heterogeneity refers to the diversity with respect to the multiple databases which make up the component databases of the multiple database system. This diversity can present as one of the following [Geo90]:

1. Diversity of hardware: configuration, instruction sets, data formats and representation (e.g., IBM mainframes, VAXes or UNISYS hardware).

2. Operating system diversity: file system, interprocess communication and transaction support (e.g. IBM/VM, VAX/VMS or UNIX).

3. Diversity in networking protocols: the networks connecting the various databases to the MDBS may have different protocols (e.g. TCP/IP, DECnet, SNA or remote procedure calls - RPCs).

4. Variations in data managers: this types of difference manifests where perhaps two component databases both use a relational database but while one uses dBase as the DBMS, another uses Access.

5. Differences in underlying data models: network, hierarchical, relational, object-oriented.

6. Transaction management protocols:

   - transaction management primitives and related error detection facilities available through the local database interfaces.
   - concurrency control, global commitment and recovery schemes used by the local database system's DBMS.

2.1.3 Autonomy

Autonomy refers to the distribution of control. It indicates the degree to which individual component databases in a multiple database system can operate independently. Breitbart *et al* [Bre95] and Özsu & Barker [Özs90] each cite three levels of autonomy. They use different terminology but basically the autonomy levels boil down to the following:

1. *Design autonomy* — no changes are to be made to local DBMS software or existing local data to accommodate the multiple database system. The local operations of the individual DBMSs are not affected by their participation in the multiple database system. Making changes to existing software may be possible but even so, modifying it is expensive and creates a major maintenance problem. Design autonomy implies that local DBMSs in a MDBS environment may use different data models and follow different concurrency control protocols, and that no modifications are made to local DBMS software. This autonomy is important since local database systems are pre-existing and may thus have followed different concurrency control protocols before their integration into the MDBS environment. Implementing a new concurrency control mechanism in all local DBMSs could degrade performance in local database systems and prove to be expensive to implement and above all would require extensive changes in the software of existing DBMSs. In cost terms, this would therefore be impractical.

2. *Execution autonomy* — each local DBMS at the database site retains complete control over the execution of transactions at its site. The manner in which the individual DBMSs process queries and optimize them is not affected by the execution of global queries that access multiple databases. An implication of this constraint is that a DBMS may abort a transaction executing at its site at any time during its execution, including the time when a global transaction is in the process of being committed by the multiple database system software layer. Even if one were to have design and communication autonomy, and not have execution autonomy, the local DBMS would not retain control over its database. It would be possible for a global transaction to hold onto locks on certain data items at a local database for an unbounded period of time. These data items would then not be available to local transactions and therefore degrade performance. This type of autonomy is especially important if participating database systems belong to different, competing organizations that may not have complete trust in one another, and wish to retain complete control over their databases.

3. *Communication autonomy* — the local DBMSs integrated by the multiple database software layer are not able to coordinate the actions of global transactions executing at several different sites. This constraint implies that the local DBMSs do not share their control information with each other or with the software layer system. This is important because each of the component database systems were built as a centralized system, and are thus unaware of any other component database systems. Also, most existing DBMSs do not communicate concurrency control information or recovery information to users. Incorporating these features and requiring local DBMSs to communicate concurrency control information may not be cost-effective and may be impractical from a software engineering point of view. Furthermore, a local DBMS which follows a locking protocol may not have serialization information readily available. Thus requiring it to provide such information may degrade performance to unacceptable levels.

It can be argued [Ras93b], that the preservation of local autonomy is both desirable and necessary in a multiple database system for the following reasons:

- Since a local or component database is essentially an independent database system, many applications have been developed prior to integration. These applications should continue to run after integration.

- Since local DBMSs controlling access to local databases had total control over their database before integration, it is desirable for them to have as much control as before, after integration.

- Local autonomy allows participating database systems to be added or removed very easily to or from a MDBS environment.

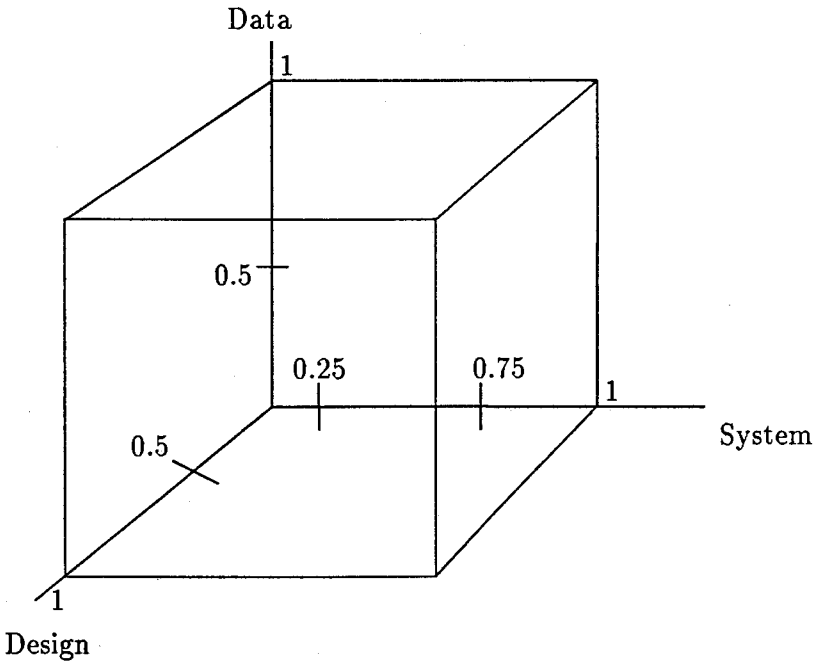


Figure 2.1: Modification dimension's axes

[Bar94, p.156]

It would be useful to have some sort of method for determining the autonomy level in specific MDBSs with specific software. We will introduce one such scheme in the following section.

2.1.4 Quantification of autonomy

Barker [Bar94] has set out a number of guidelines for the quantification of autonomy on what he calls *multidatabase systems*. We shall apply this strategy to our multiple database systems and outline his quantification methodology here in some detail for the sake of clarity. This quantification strategy will be referred to when we evaluate various protocols in Chapter 7. Barker identifies three fundamental dimensions: *modification*, *execution* and *information exchange*. The variations in these dimensions are not necessarily absolute but should be viewed as a continuum where some autonomy can be sacrificed rather than an all or nothing situation. Each dimension is awarded a value ranging from 0 to 1 — as described below — and then a method is outlined for using these values to arrive at a single value indicating the autonomy violation of a particular software protocol.

1. **Modification** — This addresses *what* needs to be modified to permit a database system's participation in the MDBS. This dimension relates to the design autonomy as identified in section 2.1.3. Figure 2.1 depicts the various aspects of this dimension.

Each axis has an equal weight, all axes have the same length. This means that each dimension which is used to quantify autonomy is equally important in the calculation of a value for autonomy violation. When no violation occurs the value zero is assigned to the axis while a maximal violation will be assigned a value of 1. The axes are:

- **The data itself** *(on the y axis)*: We look at whether changes must be made to how data is stored on local databases. This means that extra data items must be added to, or removed from, the database to fit in with other databases which are going to be components of the MDBS. The origin is where modification to the data is unnecessary. Adding new data is the next point on the axis. This may be used to provide concurrency between different DBMSs or to facilitate the mapping of heterogeneous schema's. This is a 0.5 violation. The worst violation occurs when changes must be made to existing data to permit participation in the MDBS. For this a weight of 1 is assumed.
- **The DBMS system** *(on the x axis)*: We can identify four important points on this axis, each representing a more invasive modification to the system. If the DBMS is not modified, then it is placed at the origin. Ideally no additional software would be required at the local DBMS. This is an unrealistic goal because all recovery protocols which have been devised for multiple database systems, thus far require certain changes to the local DBMSs. Thus we must consider the various possible violations of the autonomy. If additional software is required, this is not a major violation so it is placed at the 0.25 mark. Most MDBSs make assumptions about the environment and require component systems to conform to the assumption or be modified. This is a significant invasion of autonomy and is weighted at 0.75. Finally, if the number of modifications to the local systems is such that assumptions about the DBMS impose new standards on it, a major autonomy violation has occurred and such proposals will be assigned a 1 on the system modification axis.
- **The way the database is designed** *(on the z axis)*: This addresses changes to the underlying local schemas used by each DBMS. The ideal point is the origin where nothing is changed. The midpoint of the axis indicates that mapping functions are necessary to permit the data to be used by another DBMS. The most invasive point occurs when it is not possible to do a mapping and so some or other local schema has to be changed. These will inevitably cause changes to the local applications as well.

2. **Execution** — This addresses the level, either global or local, that controls the execution sequence at the individual database site. This dimension relates to the execution autonomy as identified in section 2.1.3. Autonomy measurement can be based on the component that controls local transactions and global transactions. Global trans-

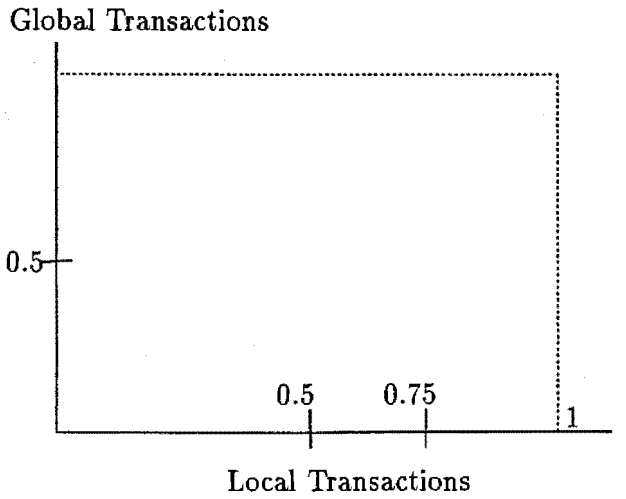


Figure 2.2: Execution dimension's axes

[Bar94, p.158]

actions are typically split up into global subtransactions — one such subtransaction for each database which is accessed by the global transaction. Global subtransactions (GSTs) can thus be seen as *agents* for global transactions which execute the part of the global transaction which accesses the data items in that particular local database system. When discussing the component that controls them, they are logically the same. Transactions carry out various operations on the database but the most important for the purpose of this discussion are *read, write, commit* and *abort.* The transaction will eventually reach a commit point and then will either commit (in which case all the operations carried out on the database become permanent) or abort (in which case none of the operations carried out on the database will be reflected on the database). Figure 2.2 illustrates how the two types of transactions are measured.

- **Local transactions** *(on the x axis)*: execute independently of the MDBS — shown by the 0 value on the axis. Where it is necessary to coordinate the execution of local transactions with other transactions through interactions with the MDBS, an autonomy violation occurs. This is indicated at the 0.5 point on the axis. If local transactions have to be submitted through the MDBS software layer instead of, or in addition to, the normal local transaction interface, then a point value of 0.75 is assigned. Finally a 1 is assigned if local transactions can only be executed by submitting them to the MDBS layer.
- If we have a look at the **global transaction** axis *(the y axis)*, the zero level is allocated to systems where the global subtransactions are submitted and treated in exactly the same way as the local transactions. The midpoint of the axis indicates that GSTs are executed to commit point and then must get confirmation

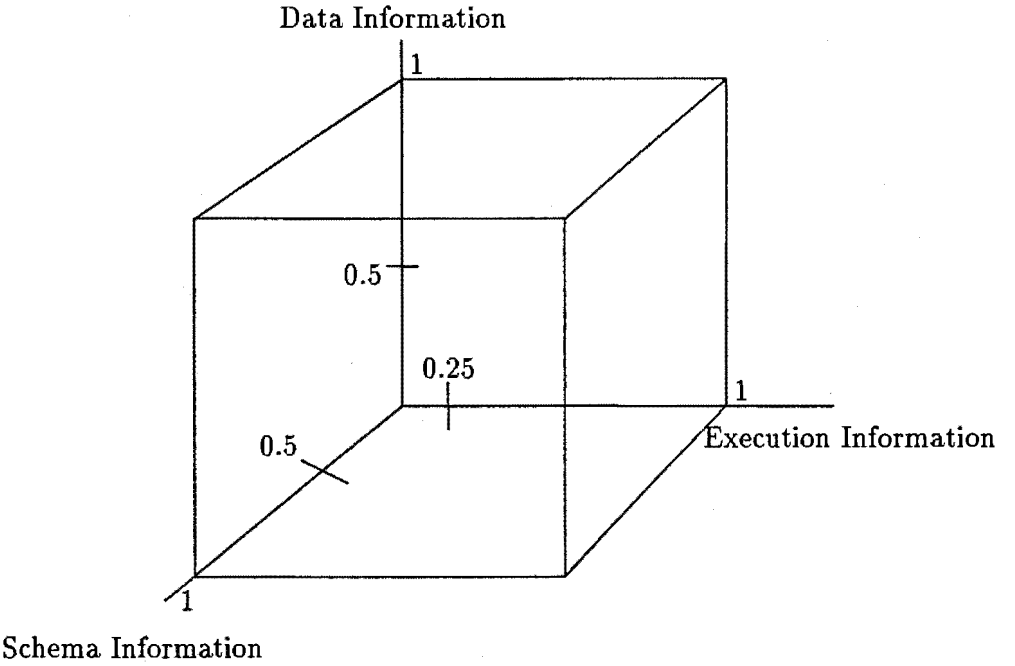


Figure 2.3: Information exchange dimension's axes

[Bar94, p.159]

from the MDBS before the transaction is committed. Finally, the most serious violation occurs when the MDBS wholly controls the execution sequence of the global transactions. In this case the local DBMS becomes simply a workhorse for the MDBS.

3. **Information exchange** — This dimension addresses the amount of information that must pass between the local and global levels to permit a database system to join or leave the MDBS and to ensure correct execution while it is functioning. This dimension relates to the communication autonomy as identified in section 2.1.3. The dimensions are illustrated in Figure 2.3. Dimensions of interest here are:

   - **Data information** *(on the y axis)*: How data is exchanged between levels. The zero on this axis indicates that all heterogeneity between data and data models is handled by the MDBS. If data from one database must be stored at another database, possibly in another format, a value of 0.5 is assigned. The most invasive approach occurs when data intended to represent a particular value must be changed to guarantee consistency over multiple databases.
   - **Schema information** *(on the z axis)*: How schema information is exchanged when a database system joins or leaves the multiple database system. The origin

represents the approach where only schema information for data that the local data wants to make available is transmitted to the MDBS. The midpoint is where additional mapping information must be exchanged by the DBMSs either directly or via the MDBS. Most invasive of all is when a DBMS must provide its entire conceptual schema so that the database system can participate in the MDBS.

- **Execution information** *(on the x axis)*: How concurrency control, deadlock and reliability information is exchanged at runtime. The origin on this axis is where no information must be exchanged between the levels during the transaction's execution. If information is exchanged then communication autonomy is sacrificed. It may be possible to exchange certain kinds of information without being too invasive. For example, if the MDBS only requests status information from the DBMS, that does not violate autonomy. In this type of situation, we use the 0.25 mark to acknowledge that a trade-off has occurred. When the DBMS must get status information from the MDBS regarding a transaction's execution, this is regarded as a severe invasion and a value of 1 is awarded.

There is some overlap between the axes of various dimensions. This is because there is a close relationship between the various autonomy dimensions as described above and should be considered beneficial in measuring autonomy because a small violation in one area may impact on other dimensions. When we have values for the three (or two) axes of autonomy violation for any of the individual dimensions, namely x, y and z, we can work out the final overall autonomy violation indicator for that dimension by using the following formula:

$$\bar{n} = \sqrt{x^2 + y^2 + z^2}.$$

The maximum value for $\bar{n}$ is $\sqrt{3} \sim 1.732$. A maximal violation of any single axis is significant because if it is maximally violated, it represents 57.7% of the maximum and a maximal violation along two axes is 81.6%. This reflects the characteristic of this quantification method which reflects maximal violations far more seriously than multiple minor violations. One also needs to make an adjustment so that the execution dimension makes the same impact on the final result as the other two dimensions because if we use the same method its maximum contribution would be $\sqrt{2}$ which is less than the maximum value of the other dimensions, namely $\sqrt{3}$. The easiest way to do this would be to multiply the final value awarded to the execution autonomy by $\sqrt{1.5}$. This permits the execution dimension to exhibit the same characteristics as the other.

After quantifying these three dimensions, we can work out a total autonomy violation taking the three dimensions: m : *modification*, e : *execution*, and i : *information* into account. Now finally we can work out an autonomy violation value, $\bar{a}$, for the entire system

as follows:

$$\bar{a} = \sqrt{m^2 + e^2 + i^2}.$$

The maximum length of each dimension is $\sqrt{3}$ so that the maximum value of $\bar{a}$ will be $\sqrt{9} = 3$. A single value therefore represents a measure of the autonomy violated by a particular system integration approach [Bar94].

Using these measurements, we can define the terms *fully autonomous, semi-autonomous* and *non-autonomous.*

- A multiple database system is said to be **fully autonomous** if the individual database systems making up the system are stand-alone database systems that know nothing about the existence of other database systems that make up the multiple database system. They also have no notion of any type of communication with the DBMSs of the other component database systems. In this case the value for the autonomy violation would be 0.

- A multiple database system is said to be **semi-autonomous** if the component database systems can operate independently but have decided to participate in a federation to make their local data shareable. They are not fully autonomous because they require certain changes to be made to their DBMSs in order to participate in the federation. The final overall autonomy violation would probably be midway between the maximum $\sqrt{9}$ and the minimum 0.

- A multiple database system is said to be **non-autonomous** if a single image of the entire database is available to any user who wants to share the information which may reside in the multiple databases. The individual database systems will typically not operate independently even though they probably have the functionality to do so.

  An example of a typical autonomy evaluation of a non-autonomous MDBS may look as follows:

```
Modification Dimension
    System          1       New standards are imposed
    Data            0       Data remains untouched
    Design          X       Not discussed
Total value of m = 1
```

```
Execution Dimension
    Local Transaction  1          Submitted via the MDBS
    Global Transaction 1          Controlled by the MDBS
Total value of e = 1.732

Information Exchange Dimension
    Execution          1          Execution coordinated by the MDBS
    Data               X          Not discussed
    Schema             1          Entire Schema provided
Total value of i = 1.414
```

The overall autonomy violation is $\bar{a} = \sqrt{m^2 + e^2 + i^2} = 2.5$.

2.1.5 Cost of autonomy

Autonomy does not come free. Four aspects can be identified that can be adversely affected by autonomy [Gar94]:

- *Correctness* — If there is a high level of transaction autonomy, the question of execution correctness can be raised. For example, one way of maintaining correctness in distributed systems is through the use of lock-based distributed concurrency control mechanisms. When a node has lock autonomy, it can release a lock acquired by a nonlocal transaction and thereby possibly violate an established locking protocol, which may breach correctness criteria.

- *Timeliness* — With autonomous setting of priorities for nonlocal transactions, no guarantees can be made about how soon such requests will be serviced. This could cause a global request to have an unacceptably long response time or to be starved altogether.

- *Level of cooperation* — The issue of cooperation involves data load sharing among nodes in a distributed system. When cooperation is mandatory, node autonomy is very difficult to maintain.

- *Degree of data replication* — High autonomy almost certainly implies that data will be replicated because we will essentially be integrating pre-existing databases with their own data contents. Because of nodal autonomy, this replication will have to be maintained. On the one hand, the replication eases scheduling, name translation and execution autonomy, but it also carries with it the problems of ensuring data consistency among replicated data items.

2.2 Classification of Multiple Database Systems

There seems to be much confusion in the literature about the classification of multiple database systems. Three main classification methods have been identified; firstly classification according to architectural alternatives [Bel92], secondly classification according to degree of autonomy, heterogeneity and distribution [Özs90] and thirdly according to how tightly the participating multiple databases are coupled in the resulting system [Bri92]. A combination of these methods resulted in the taxonomy of multiple database systems described below and illustrated in Figure 2.4.

The levels of the classification firstly consider distribution, followed by heterogeneity and autonomy at the lowest level. Within each of these sublevels the different architectural alternatives are used as differentiators.

2.2.1 Non-distributed multiple database system

If the multiple databases are not distributed over a number of sites, the system is referred to as a **Non-distributed multiple database system.** It can also be called a *logically integrated multiple database system.* There are not many examples of this type of system, but this type of system could be suitable for a system consisting of multiprocessors where all the databases reside at one site and are accessed by all the users in the system.

Within this main group we can further distinguish between heterogeneous and homogeneous database systems.

1. **Heterogeneous Non-Distributed Integrated Multiple Database System:** In this type of system one has multiple databases which are heterogeneous with respect to database structure, data managers or other aspects, but which provide an integrated view to the user. This type of system, for example, could provide access to a network, hierarchical and relational database, all residing on the same machine.

    - **Single Site Heterogeneous Federated Database System** — In this type of system the non-distributed, heterogeneous databases are *semi-autonomous.* This could be a system where the various existing databases at a single site must still be used individually but they also form part of a multiple database system for another set of users who require access to all the available data at the site. The individual databases will probably have to have certain alterations made in order to participate in the multiple database system.
    - **Heterogeneous Unaffiliated Multiple Database System** — In this type of system non-distributed, heterogeneous database systems are *fully autonomous.* This means that the individual databases and their data managers will not have any changes made to them in order to participate in the multiple database system.

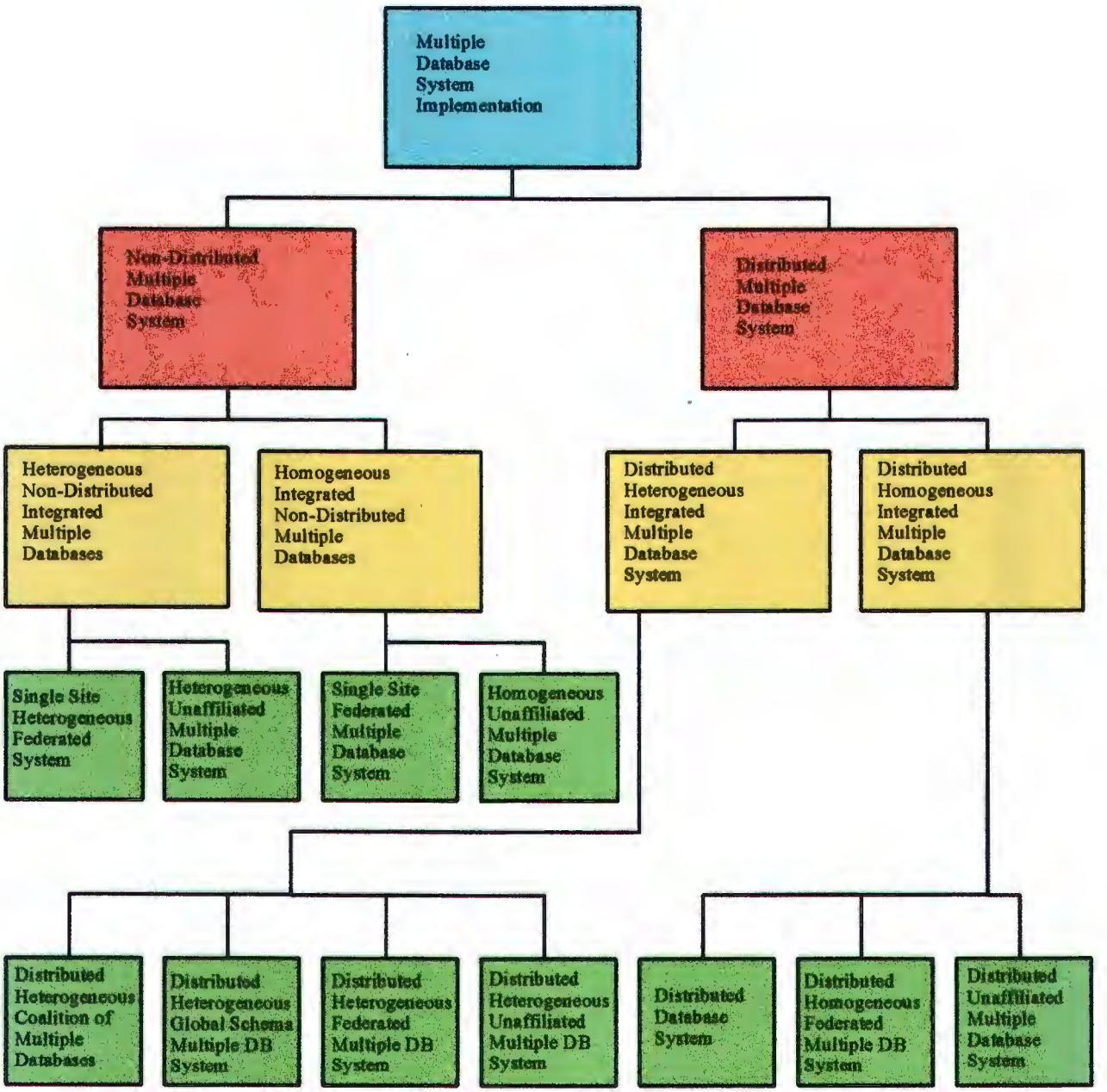


Figure 2.4: Classification of multiple database systems

The databases will all reside at the same site, will be heterogeneous and because of the full autonomy, will see the multiple database system software layer as simply another user.

2. **Homogeneous Integrated Non-Distributed Multiple Database System:** This multiple database system would typically consist of a set of databases which all have the same data managers and the same underlying structure all residing at the same site.

    - **Single Site Federated Multiple Database System** — In this type of system, the non-distributed homogeneous multiple databases are *semi-autonomous.* Thus certain changes may be made in order to reconcile differences between data items stored in different databases, or changes could be made to transaction management methods in the local DBMS.
    - **Homogeneous Unaffiliated Multiple Database System** — In this type of system the non-distributed, homogeneous, multiple databases are *fully autonomous.* In this architecture, no changes would be made to individual databases and the multiple database software layer would have to reconcile any differences between the databases before doing queries or updates on the databases.

2.2.2 Distributed multiple database system

If, however, the multiple databases are distributed, the system is referred to as a **distributed multiple database system.** In this type of system the database is distributed over various sites even though an integrated view is provided to the users. A communication medium will be used to communicate with the component database systems.

Within this main group we can once again further distinguish between heterogeneous and homogeneous database systems.

1. **Distributed Heterogeneous Integrated Multiple Database System** — In this type of system the component database systems are heterogeneous. As the name suggests, the component databases in this system are distributed over various sites and would have various differences between them and/or their data managers.

    - **Distributed Heterogeneous Coalition of Multiple Database Systems:** In this type of system, also called a tightly-coupled system, the component database systems would have no autonomy whatsoever, thus *non-autonomous*, and would probably not function independently even though they have the functionality to do so. The global system has total control over local data and processing. The system will create a global schema by integrating the schemas of all the participating multiple database systems. A single image of the database would be

made available to any user who wants to use the data in the multiple databases. From the users' perspective, the data is logically centralized in one database.

This type of system is described by Özsu *et al* [Özs90] as a *tightly integrated multidatabase management system.* Because it is so tightly integrated, this type of system can closely synchronize global processing. Also, since the global system has complete control over local systems, processing can be optimized for global requirements. These systems have good global performance but this is achieved at the cost of significant local modification and loss of control [Bri92].

If our pharmacy example were to be implemented this way, it would mean that all transactions would have to be entered by the new owner — the global user. There would be no local users in the system at all.

- **Distributed Heterogeneous Global Schema Multiple Database System:** This type of system integrates *semi-autonomous* multiple databases. These types of systems are more loosely coupled than the coalition of multiple database systems described above, because global functions access local information through the external user interface of the local DBMS. However, the multiple database system software layer still maintains a global schema, also called the global conceptual schema, so the local sites must cooperate closely to maintain the global schema. These systems are typically designed bottom-up and can integrate pre-existing multiple databases without modifying them. This class of multiple database systems is introduced by [Bri92].

  Creating the global schema here is more difficult than in the coalition of multiple database type of system because the DBA(Data Base Administrator) of the global system has no control over local schema input to the global schema and the local schemas are not modified when they join the global schema multiple database system.

  In this type of multiple database system, we have three extra levels on top of the ANSI-SPARC architecture (see Figure D.1, Appendix D):

  - The *global conceptual schema* — In this type of system the global conceptual schema is simply a logical view of *all* the data available to the multiple database system. It is only a subset of the union of all the local conceptual schemas, since the local DBMSs are free to decide what parts of their local databases they wish to contribute to the global schema.
  - *The participation schema* — A component database's participation in the multidatabase system is defined by means of a participation schema and represents a view defined over the underlying local conceptual schema [Bel92]. It represents the extent of the local database's participation in, or contribution towards, the multiple database system.

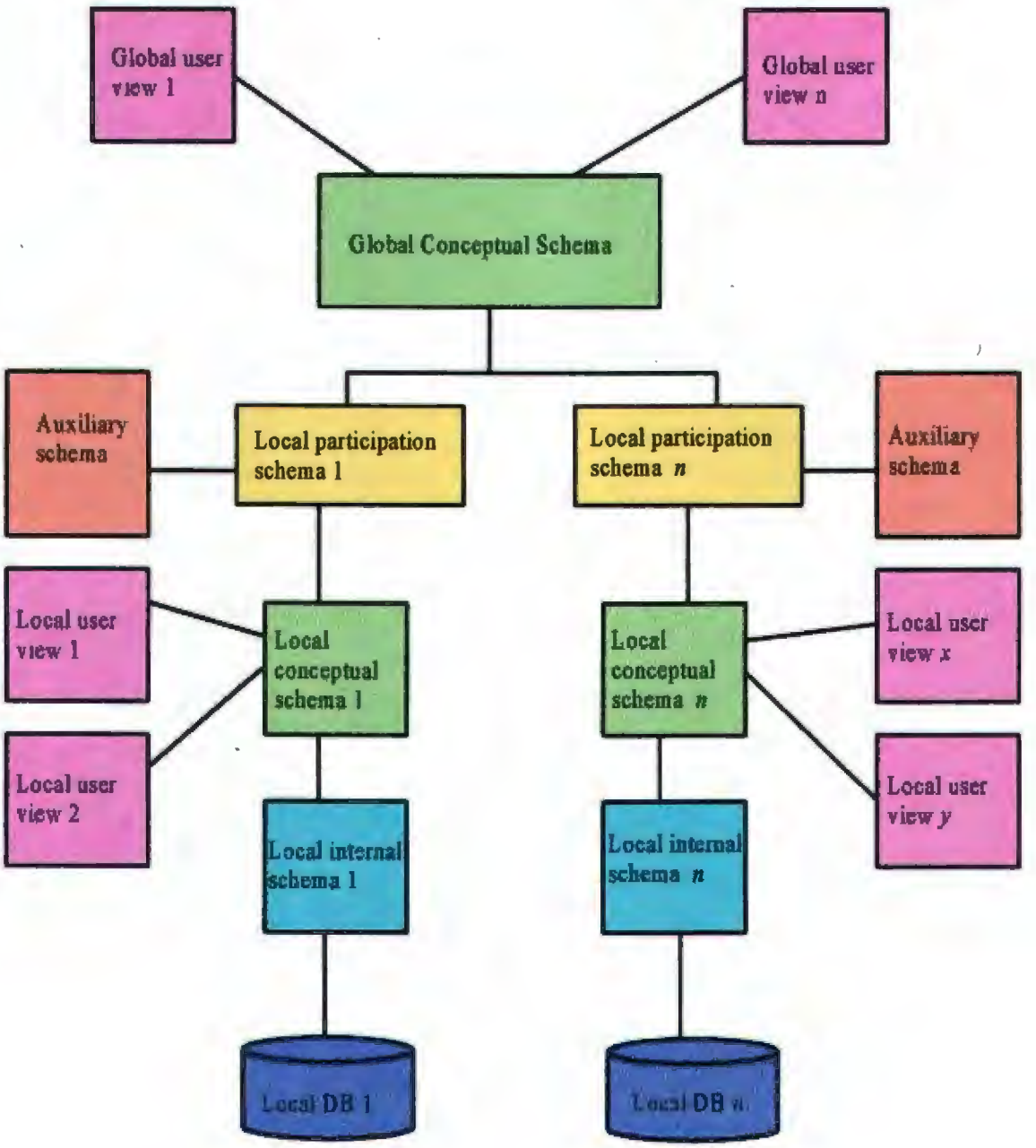


Figure 2.5: Schema architecture of a distributed heterogeneous global schema multiple database system

[Bel92, p.50]

- *The global external views* — Support for user views, which is very important because the global conceptual schema will probably be extremely large, is provided by the auxiliary schema. The auxiliary schema, illustrated in Figure 2.5, describes the rules which govern the mappings between local and global levels. For example, rules for unit conversion may be required when one site expresses distance in kilometres and another in miles. Rules for handling null values may be necessary where one site stores additional information which is not stored at another site, for example one site stores the name, home address and home telephone number of its employees, whereas another just stores name and address.

Some multiple database systems also have a fragmentation schema although not an allocation schema since the allocation of fragments to sites is already fixed as multiple database systems integrate *pre-existing* databases.

- **Distributed Heterogeneous Federated Multiple Database System** — In this type of system, component database systems are also *semi-autonomous* but in this type of system there is no global schema. The component database systems typically sacrifice some of their autonomy to become part of the MDBS. This would cause some additional difficulties in allowing databases to join or leave the system but on the other hand, would make concurrency and recovery maintenance easier to enforce.

  These systems — illustrated in Figure 2.6 — are once again more loosely coupled than the global schema multiple database system. Each local system maintains a *local import* and *export schema.*

  - The *export schema* is a description of the information the local node is willing to share with the global system.
  - The *import schema* is a description of the information from the other database systems that may be accessed locally. Each import schema is therefore a partial global schema.

  Therefore, each local participating database system must cooperate closely only with the nodes it accesses in order to carry out some transaction. This is done by means of the import schema provided by the remote database system that needs to be accessed. User queries are restricted to local data and the data represented in the local import schema. [Bri92]

  This would probably be the best option for our pharmacy example multiple database system because the component database systems would retain their functionality while allowing the global user access to most of the data as well.

- **Distributed Heterogeneous Unaffiliated Multiple Database System** — In this type of system the multiple database systems are *fully autonomous.* Özsu

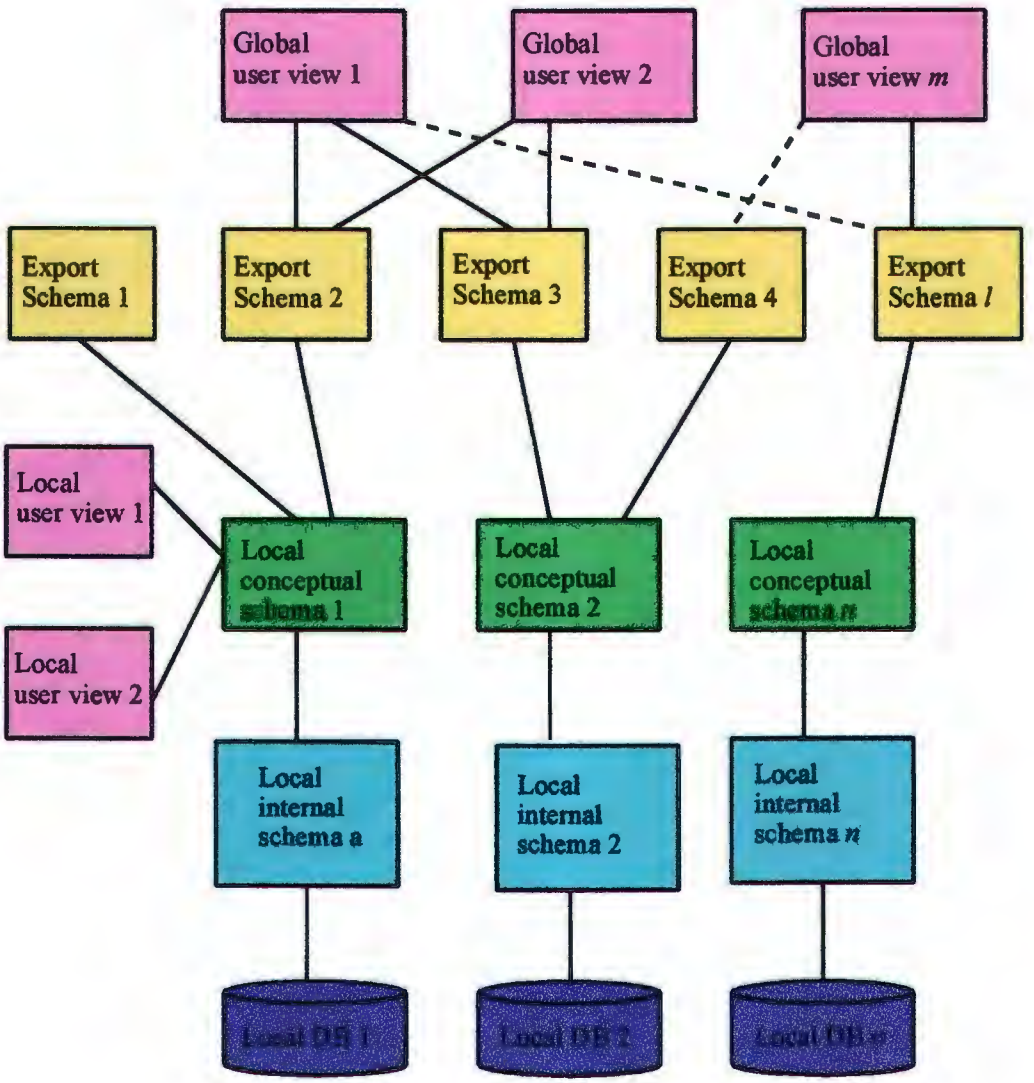


Figure 2.6: Architecture of a distributed heterogeneous federated multiple database system [Bel92, p52]

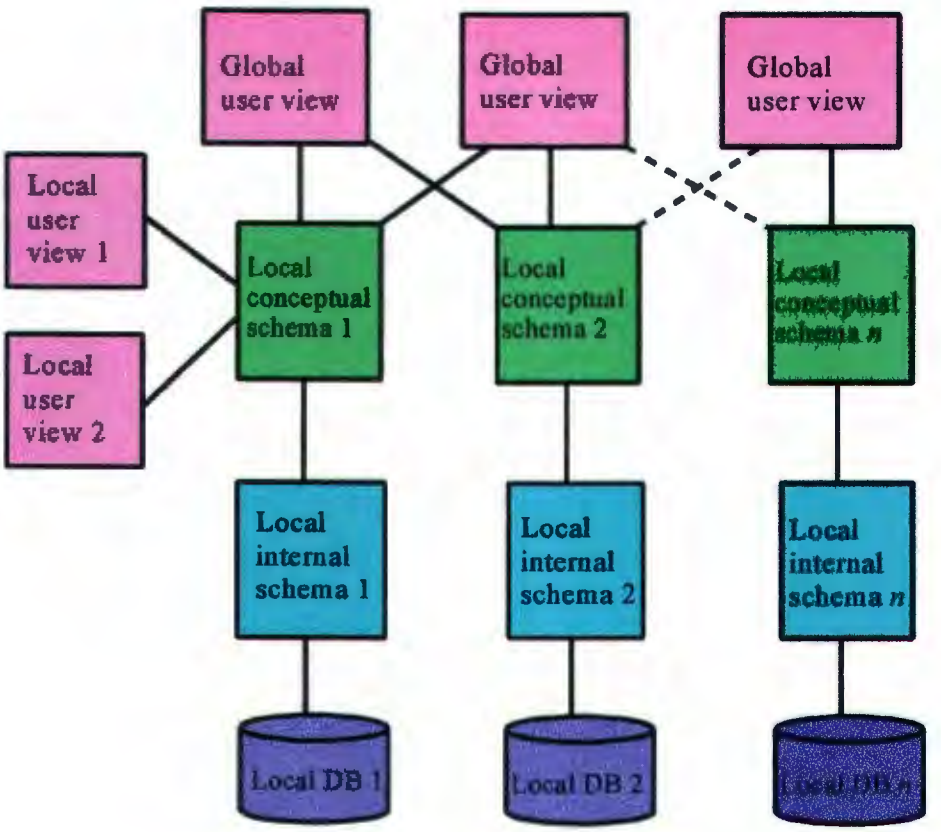


Figure 2.7: Unaffiliated multiple database system architecture

[Bel92, p.51]

et al refer to this system as the *total isolation* option [Özs90]. This means that absolutely no changes are made to the DBMSs of the component database systems in order for them to be part of the multiple database system.

This option is almost impossible to achieve if we want any update functionality at all. In this type of system, the processing of user operations is especially difficult since there is no global control over the execution of transactions by individual DBMSs [Özs90]. The components of this type of system are illustrated in Figure 2.7.

In these systems no global schema is maintained. The global system supports all global database functions by providing query language tools to integrate information from separate databases in the multiple database system. User queries can specify data from the local schema of any of the individual databases participating in the system [Bri92].

The construction of a global conceptual schema is a difficult and complex task and involves resolving both semantic and syntactic differences between sites [Geo90]:

- *Semantic* differences relate to the meaning and intended use of data. These include inconsistencies in the domain definitions for attributes, discrepancies in naming (synonyms and homonyms), data values and value precision.
- *Syntactic* differences include differences in data types used (e.g. identity number can be defined as a string or an integer), data manipulation operators (e.g. the set of operations in the CODASYL model which is equivalent to a join operation in the relational model) and units of measurement (e.g. feet or metres).

Sometimes these differences are so extensive that it does not warrant the huge investment involved in developing the global schema, especially when the number of multisite global queries is relatively low [Bel92]. Several researchers argue that this type of organization gives this system a significant advantage over non-autonomous type multiple database systems. The unaffiliated multiple database system typically has two layers, the *local database system layer* and the *multiple database system layer* on top of it. The local system layer consists of the component database systems. The responsibility for providing access to multiple heterogeneous databases is delegated to the mapping between the external schemas and the local conceptual schemas. This is fundamentally different from the global schema multiple database option, where this responsibility is taken over by the mapping between the global conceptual schema and the local ones.

This difference in responsibility has a practical consequence. If a global schema is defined, then the consistency rules of the global database can be specified according to this single definition. If a global schema does not exist, however, dependencies have to be defined between the various local conceptual schemas [Özs90].

2. **Distributed Homogeneous Integrated Multiple Database System** — Because the component database systems are homogeneous, the task of the multiple database system software layer is simplified a great deal. It has none of the problems associated with heterogeneous systems. Our pharmacy example does not fit into this category because the component databases are not homogeneous.

   - **Distributed Database System** — In this system, the component databases have *no autonomy* at all. The traditional distributed database concept falls into this category. These systems are typically designed in a top-down fashion, with local and global functions implemented simultaneously. The same functional interfaces are presented at all levels even though they may be implemented on different machines. The global system has control over local data and processing. Global users access the system by submitting queries over the global schema.

They closely synchronize global processing because they are so tightly integrated. Processing can also be optimized very well. This type of database system usually performs well but this is achieved at the cost of significant local modification and loss of control. [Bri92] To formally define this concept:

> A **distributed database** (DDB) can thus be defined as a *logically integrated collection of shared data which is physically distributed across the nodes of a computer network* [Bel92].

while a distributed database management system can be defined as:

> A **distributed database management system** (DDBMS) is a multiple database system software layer which manages a distributed, homogeneous, non-autonomous or autonomous set of multiple databases in such a way that the distribution aspects are transparent to the user.

- **Distributed Homogeneous Federated Multiple Database System** — In this system the distributed, homogeneous, component databases are *semi-autonomous.* Because the component database systems are homogeneous, the component database systems are the same, and it is a lot simpler to integrate them into a federated type system. Certain changes may be required, for example in the recovery protocols of the various systems, in order to make them part of the federation. The component database systems will still function independently however. The structure of this system is the same as the distributed heterogeneous federated multiple database system.

- **Distributed Homogeneous Unaffiliated Multiple Database System** — In this system, the component database systems are *fully autonomous*, distributed and homogeneous. Özsu *et al* also refer to this system as the *total isolation* option [Özs90]. The organization of this type of system as well as its management is quite different from that of a traditional distributed database system. The fundamental difference lies in the difficulties caused by the autonomy of component data managers. The structure of this system is the same as the distributed heterogeneous unaffiliated multiple database system.

2.3 Multidatabases

2.3.1 Definition

In some of the literature, full autonomy, distribution and heterogeneity is required when a multidatabase system is referred to. For the purpose of this dissertation, we are going to include the semi-autonomous, distributed, heterogeneous multiple database systems in our general definition of a multidatabase. A *multidatabase* can thus be defined as:

> **Multidatabases** (MDB) are systems which are composed of autonomous or semi-autonomous, heterogeneous, distributed, *pre-existing* databases, together with a software layer, built on top of them, to control access to all data in all the component databases from the point of view of the user of the multidatabase system, while the component database systems still function independently.

> A **Multidatabase Management System** (MDMS) is the software layer built on top of the multiple database systems that facilitates access and manipulation of data at local sources (the component databases), distributed among nodes of a computer network by both users at the local databases as well as users of the multidatabase system.

This would then include the following categories:

- Distributed Heterogeneous *Global Schema* Multiple Database Systems,
- Distributed Heterogeneous *Federated* Multiple Database Systems,
- Distributed Heterogeneous *Unaffiliated* Multiple Database Systems.

2.3.2 Reasons for using the multidatabase concept

There has been much research into the integration of information resources. The main objective of multidatabase systems is to provide organized access to multiple database systems. In this section we look at the motivation behind the development of the multidatabase concept.

The demand for a multidatabase arises in response to the need for an integration mechanism for an organization which has a large number of operational database systems that are already in use and support their own databases, applications and users. The organization cannot simply create a distributed database system here because of the investment which went into developing these individual systems. The need for multidatabases has also emerged in public networks which provide access to many different types of databases. These databases are often owned by a variety of different people or organizations and provide a

vast variety of services. The motivation behind not integrating these types of databases into a single database, can be summarized as [Geo90]:

- *User autonomy* — Users have different needs and prefer to represent their data in different ways. Instead of providing a database schema and a DBMS to satisfy all needs, users with similar requirements can be accommodated far more effectively by allowing them to design their own databases to suit their own needs.

- *Ownership and security considerations* — In a large organization, one often has data which belongs to some or other department in the organization. Other groups may need occasional access to the data but the group which creates and has frequent access to the data, should control it. The multidatabase approach allows this facility.

- *Unique features provided by a particular DBMS* — Depending on the data in the database, different features can be required. For instance, transactions in a business database usually have a short lifespan while transactions in a technical database often take hours to perform. Technical DBMSs also require additional functionality not required in a business database. These needs can be met in specific DBMSs but are almost impossible to achieve in an integrated system.

- *Reliability, availability and flexibility* — In a multidatabase system, the loose coupling of component database systems increases the overall system reliability and availability. It also allows flexibility to add and remove individual databases as the need arises.

We need a facility to access these multiple databases. Multidatabase systems attempt to solve this type of difficulty.

2.3.3 Transactions and users in multidatabases

In multidatabases, the component databases will be referred to as the local databases and their users will be referred to as local users. In our example, the databases in Pretoria, Port Elizabeth and Cape Town are the local databases and the users at each of these sites are the local users.

However, a new dimension has been added. We now also have a new set of users, the *global users.* These users access all data incorporated into the MDB which resides in the local component databases, via the MDMS. These users are controlled by the MDMS data administrator and are unknown to the individual local component databases. The MDB will accept transactions from these users, split them up into subtransactions and submit them to the applicable databases as transactions. To the local database, the transaction that comes from a local user and the transaction coming from the MDMS level, looks exactly the same, and is treated identically. This means that while each local database has facilities to execute transactions, each individual local database has no notion of executing distributed transactions that span multiple components (because they have neither global

concurrency control mechanisms nor distributed commit protocol implementation).

In summation: a MDMS thus supports two types of transactions and users. Transactions are divided into:

- *Local transactions* that access data at a single local site outside of the MDMS control. These transactions result from the execution of user programs submitted directly to the local DBMS.
- *Global transactions* that are executed under the MDMS control. These result from the execution of user programs submitted to the MDMS. A global transaction consists of a number if subtransactions, each of which is an ordinary local transaction from the point of view of the local DBMS where the transaction is submitted.

There are also both local and global users in a MDMS:

- *Local users* can continue to access their databases in the normal way unaffected by the existence of the MDMS. Each local DBMS has its own transaction processing components, including a concurrency control mechanism that ensures serializable and deadlock-free execution of all transactions submitted, both from local and global users [Bre95].
- *Global users* which access the individual databases only via the MDMS layer. From the perspective of each DBMS, the MDMS layer is simply another *'user'* from which they receive transactions and present results. The only type of communication between the autonomous database systems is via the MDMS layer.

The terms *local* and *global* has been used throughout this dissertation in order to distinguish between aspects which refer to a single site (local) as contrasted to those aspects which refer to the system as a whole (global). The global database is a virtual concept as it does not exist physically anywhere.

If we refer back to our pharmacy example, the pharmacist and his/her assistants at each individual pharmacy are the local users while the new owner of the pharmacies will use the data as a global user via the MDMS layer. The global transaction will be submitted to the local DBMS as if it comes from simply another user and the results of the transaction will be sent to that user - thus the MDMS layer.

The scheduling of the global transactions is done at the MDMS layer by the *global transaction manager* (GTM) [Özs90]. The objective of MDMS transaction management is to ensure multidatabase consistency in the presence of local transactions.

2.3.4 Management of heterogeneous distributed multidatabase systems

Data in the multiple database system is handled by the global data manager (GDM). The GDM performs both the mappings between the global view of the data and the local DBMSs

and all the relevant I/O operations [Gli84].

- *Input* — The initial input to the GDM is either a query or transaction formulated on the global schema, global external views or the export schemas. If the query or transaction is directed to an individual DBMS, then it is translated into the local query language and passed to the local database system. If a distributed query or transaction is formulated, the GDM transforms the original query or transaction into a collection of subqueries, each in a format acceptable to one of the local DBMSs.

- *Output* — The GDM generates a plan of subquery execution and passes these to the multiple database systems. At each of the multiple database systems the subquery will be presented to the appropriate DBMS. The local DBMS will send the results of the query or transaction back to the GDM and the GDM will assemble the results of all the subqueries and produce the answer to the original query.

- *Functions* — A GDM must include the following five functions:

  - Global data model analysis — If the multiple database system has a global schema, there is a unified view of all the data in the multiple database system. If there is no global schema, we would still have the export schemas of each local database system and that would be used in order to set up a data dictionary which would be consulted by the GDM in order to do the mappings between user queries and data held in the multiple databases.

  - Query decomposition — the query decomposer takes the original query and fragments it into subqueries. In order to divide the global query into its subquery components, the GDM uses the distributed data dictionary as a guide. A global query which references only a single local database system does not need to be decomposed since the entire query can be carried out at a single site.

  - Query translation — this is a language to language translation which takes into account the underlying data model differences. The user would be assumed to issue the query in the unified global query language. The translator may receive the original query that is formulated on the data available throughout the multiple database system, and translate it to a global query that is based on the global schema of the multiple database system (or the union of the external schemas provided by the multiple databases). Then the translator would have to translate a subquery, for each database system that has to be accessed, into the language used by that DBMS. The translated subquery is then sent to the local database system.

  - Execution plan generation — this part of the GDM interacts with the network by passing the subqueries generated by the GDM to each individual database system. The execution plan generator decides which subqueries can be sent in

parallel, which subqueries must precede others and what the relationships are among intermediate results.

- Results integration — the results of the subqueries are combined by the results integrator which then combines them and represents the results in a form acceptable to the original user.

These five functions are basic features of a GDM in both heterogeneous and homogeneous systems although the query translation and results integration are far more elaborate in heterogeneous systems.

Following this description of what is required of the data manager in a multidatabase system, we can now delineate the functionality of a multidatabase management system.

2.3.5 Functionality of an MDMS

We have spent some time defining exactly what a multidatabase is, now we can set out exactly what the functionality of a multidatabase management system should be. To return to the problems identified initially with centralized database systems and the need for autonomy of individual databases, we can now explain how the multidatabase concept tends to solve the problems and provide the means to satisfy user needs while still providing access to all the data by global users. The overall goal of a MDMS is to ease the burden of the application programmers by providing a layer of integrating and coordinating services, acting as a front-end to individual component databases. Ideally, the user is presented with a single uniform view of a virtual database and is unaware of the autonomy, heterogeneity and distribution of the underlying data sources.

In order for the MDMS to do this, the MDMS must provide support for schema integration and management, query optimization and processing, transaction management and security [Tan93]. To elaborate:

- The MDMS should provide an *integrated view of the data* needed by an application or a group of applications. Some MDMSs assume that a global schema will be available while others have no global schema. A partial schema is also a possibility. However, the MDMS must use what is available and perform schema translation, integration and management functionalities.

- The MDMS must develop a *global query plan* and then perform the processing required for a given query. A global query plan consists of a set of component queries against individual databases as well as a data integration plan. The data integration plan will specify how to integrate the results obtained from the individual databases to produce the final result.

- The MDMS must provide *full support for global transactions.* [Tan93] contends that the autonomy and heterogeneity of the component database systems cause great dif-

ficulties in transaction management in multidatabase systems. Özsu *et al* [Özs90], however, make a convincing argument for discounting the heterogeneity aspect since it can only introduce slight additional difficulties. They conclude that the real tricky issue is that of autonomy.

- The MDMS must guarantee the *security of the data* made available through the MDMS. Each component database handles its security differently so this function is not that easy to provide. This aspect has enjoyed very little attention from researchers up to now.

Distributed multidatabase systems thus share the problems of DDBMSs and introduce additional ones of their own, such as the problem of how to split up transactions into subtransactions, how to maintain the consistency of local databases in the face of heterogeneity, and how to produce integrated results. [Özs94, Bel92, Bre95, Geo90]

It is very important to note that Özsu and Barker [Özs90] state that fundamental issues related to multidatabase systems can be investigated without reference to their distribution or heterogeneity. The additional considerations that distribution brings being no different than those of logically integrated distributed database systems for which many solutions have already been developed. Furthermore, heterogeneity need not cause any additional difficulty from the point of view of database management. The only heterogeneity aspects which we need to consider are those mentioned in item 6 in section 2.1.2 and this refers to heterogeneity of transaction management which is what we have been evaluating in this dissertation. We can thus state unequivocally that the most important issue is that of autonomy. Therefore, when discussing various alternative systems in the following chapters, we will always evaluate them with the autonomy perspective in mind.

2.4 Summary

The concept of multiple database systems and multidatabases has been introduced. In section 2.2 a classification scheme was introduced that can be used for the precise definition of concepts and terms related to multiple database systems. I found during the course of my research that there were basically three classification methods, one which classified multidatabases according to architectural differences [Bel92], another which classified them according to degree of autonomy, heterogeneity and distribution [Özs90] and yet another which classified them according to how tightly the participating local database were coupled [Bri92]. I decided to integrate the methods in order to arrive at a classification method which took all these aspects of multidatabase implementations into account and came up with the classification method presented in this chapter.

The specific multiple database system called a multidatabase has been placed in the classification taxonomy. The autonomy dimension has been identified as the characteristic which is the most important distinguishing feature of these systems and a quantifica-

tion method has been presented for measuring this dimension. Now that the background has been given, transaction management concepts of concurrent transaction processing in databases in general and later more specifically in multidatabases can be discussed.

Chapter 3

Concurrent Transaction Processing

This chapter introduces concurrent transaction processing concepts as applicable to a single database with its accompanying DBMS. The basic precepts of serializability theory are presented and various concurrency control mechanisms are outlined and illustrated by means of examples. The next chapter will show how these principles can be applied to concurrent transaction processing in multidatabases.

3.1 Introduction to Transaction Processing

Each DBMS has certain components which allow it to handle transactions [Els94]. These components are:

- a local transaction manager(LTM),
- a local scheduler(LS) and
- a local recovery manager(LRM), also called a local data manager(LDM) [Bar91].

The function of the LTM is to interface with the user and guarantee the atomic execution of transactions. The local scheduler ensures the correct execution and interleaving of all transactions submitted to the LTM. Finally, the LRM ensures that the local database contains the effects of all committed transactions and none of the effects of uncommitted ones.

When talking about transaction processing concepts, database systems can be classified according to the number of users who generally use the system concurrently. A DBMS is *single-user* if at most one user at a time can use the system and it is *multi-user* if many users can use the system concurrently. While single-user systems are usually restricted to microcomputer platforms, most other DBMSs are multi-user systems. In a multi-user DBMS, the stored data items are the primary resources that may be accessed concurrently

by user programs. These programs can either retrieve or modify the contents of the data items.

> The *execution of a program* that accesses or changes the contents of a database is called a **transaction** [Els94].

3.2 Transaction and System Concepts

The concept of an atomic transaction is fundamental to many techniques for concurrency control and recovery from failures. As mentioned before, the execution of a program that includes database access operations is called a database transaction, or simply a transaction. If the operations in the transaction do not perform any update operations, it is called a *read-only transaction*. We will use the term transaction to refer to a transaction which *does* do update operations in the database.

3.2.1 Transaction states

A transaction is an atomic unit of work that is either completed entirely or not done at all. The system needs to keep track of when the transaction starts, terminates, commits or aborts. Thus the recovery manager keeps track of the following database operations [Els94]:

- *Begin-transaction*: This marks the beginning of transaction execution.
- *Read* or *Write*: These specify read or write operations on the database items that are executed as part of a transaction.
- *End-transaction*: This specifies that read and write transaction operations have ended and marks the end limit of the transaction. Next a check needs to be carried out to see whether the changes made by the transaction can be made permanently to the database (committed) or whether the transaction needs to be aborted because it violates concurrency control requirements.
- *Commit-transaction*: This signals a successful end of the transaction so that any changes executed by the transaction can be safely committed to the database and will not be undone.
- *Rollback* (or *Abort*): This signals that the transaction was ended unsuccessfully so that any changes or effects that the transaction may have applied to the database must be undone.

In addition to these states, recovery procedures require the following additional operations:

- *Undo*: Similar to rollback except that it applies to a single operation rather than a whole transaction.

- *Redo*: This specifies that certain transaction operations must be redone to ensure that all the operations of a committed transaction have been applied to the database.

3.2.2 The system log

To be able to recover from transaction failures, the system maintains a *log*. The log keeps track of all transaction operations that affect the values of database items. The log is kept on disk and so is not affected by any failure except disk or catastrophic failures. The log is often backed up onto tape to guard against disk failure. The log entries are enclosed in [ ] and written in bold to distinguish them from transaction operations. Types of log entries are [Els94]:

[start-transaction, T**]** : Records that transaction T has started execution.

[write-item, T**,** X**,old-value, new-value]** : Records that transaction T has changed the value of database item X from old-value to new-value.

[read-item, T**,** X**]** : Records that transaction T has read the value of database item X.

[commit, T**]** : Records that transaction T has completed successfully, and affirms that its effect can be committed to the database.

[abort, T**]** : Records that transaction T has been aborted.

Some protocols do not require read operations to be written to the log. Often only the other log entries mentioned above are required as it makes the log smaller and simpler to maintain. Using the log to recover from failure is the job of the recovery manager and has been discussed in greater detail in Chapter 6.

3.2.3 Commit point of a transaction

A transaction T reaches its *commit point* when all its operations that access the database have been executed successfully and the effect of all the transaction operations on the database have been recorded in the log. Beyond the commit point, the transaction is said to be committed, and its effect is permanently recorded in the database. The transaction then writes an entry **[commit,** T**]** into the log. If a system failure occurs, we search back in the log for all transactions T that have written **[start-transaction,** T**]** in the log but not **[commit,** T**]** entry yet; these transactions may have to be rolled back to undo their effect on the database during the recovery process. Transactions that have written their commit entry to the log must also have recorded all their write operations in the log so their effect on the database can be redone from the log entries [Els94].

Note that the log must be kept on disk. Because writes to disk are buffered, the system log is always force-written at commit point so that if there is a system crash the effects of the transaction will be recorded on permanent memory [Els94].

3.2.4 Checkpoints in the system log

Another type of log entry is called a *checkpoint* [Els94]. A checkpoint record is written to disk periodically at that point when the system writes to the database all the effects of the write operations of committed transactions. Hence, all transactions that have their [**commit**, T] entries in the log before a [**checkpoint**] entry need not have their write operations redone in case of a system crash.

The recovery manager decides when to take a checkpoint. It may be done every n minutes, or after a number of committed transactions have taken place. Taking a checkpoint involves doing the following:

1. Suspend execution of transactions temporarily.
2. Force-write all update operations of committed transactions from main memory buffers to disk.
3. Write a [**checkpoint**] record to the log, and force write to disk.
4. Resume executing transactions.

The checkpoint record in the log may also include additional information, such as the list of active transaction identifiers, and the locations of the first and most recent records in the log for each active transaction. This can make it easier to undo transactions that have to be rolled back at a later stage.

3.2.5 Desirable properties of transactions

Atomic transactions should possess several properties [Els94, Bel92, Özs91]. These are often called ACID properties (Atomicity, Consistency, Isolation, Durability), and they should be enforced by the concurrency control and recovery methods of the DBMS. The following are the ACID properties:

- *Atomicity*: There are two aspects of transaction atomicity [Pu91b]:
  - Recovery atomicity: this means that when a transaction is executed, it is either executed in its entirety or does not have any effect whatsoever on the database. It is the responsibility of the recovery method to ensure recovery atomicity. If a transaction fails to complete for some reason, the recovery method must undo any effects of the transaction.
  - Concurrency atomicity: This means that users are assured that concurrent execution of another transaction will not affect their own transaction.
- *Consistency preservation*: The consistency of a database is simply its correctness. A correct execution of the transaction must take the database from one consistent state to another. The consistency preservation property is generally considered to be

the responsibility of the programmers who write database programs. The transaction should execute in such a way that if the database was consistent before execution of the transaction then it will be consistent thereafter as well, assuming no other transaction interferes.

- *Isolation*: A transaction should not make its updates visible to other transactions until it is committed; this property, when enforced strictly, solves the temporary update problem and makes cascading rollbacks of transactions unnecessary[1]. Isolation is enforced by the concurrency control method.

- *Durability or permanency*: Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure. This property is the responsibility of the recovery method.

3.3 Transaction Execution

In this section a formal transaction model based on the work of [Ber87, Özs91, Bar91, Bel92] has been outlined. This model makes it easier to reason about the transaction concepts that apply to concurrency control and recovery in database systems. First of all, the basic transaction operations need to be formally defined.

3.3.1 Basic transaction operations

A DBMS supports various commands that may be used to access data items in a database. These are called *operations*. The database access operations that a transaction can include are:

- *read-item(X)*: Reads a database item named X into a program variable. We assume that the program variable is also called X.

- *write-item(X)*: Writes the value of program variable X into the database item named X.

Definition 3.1 — *Database operations*

We denote by $O_{ij}(X)$ some operation O_j of transaction T_i that operates on database entity X. $O_j \in \{read\text{-}item, write\text{-}item\}$.

OS_i denotes the set of all the operations in T_i.

[Özs91] □

[1] See section 3.5

The *read-item* and *write-item* operations will be denoted by r and w, respectively, for the rest of this dissertation. We have also used the convention of using calligraphic lettering to denote formal sets and roman fonts for acronyms. Thus in the following definition, BS is an abbreviation for "base-set" while $\mathcal{BS}$ denotes the set of data items in the base set of a transaction.

Definition 3.2 — *Read-set(RS), Write-set(WS) and Base-Set(BS)*

The set of data items that a transaction reads are said to constitute the *read-set*($\mathcal{RS}$). Similarly, the set of data items that a transaction writes are said to constitute its *write-set*($\mathcal{WS}$). The read-set and the write-set need not be mutually exclusive. Finally, the union of the read-set and the write-set is called the *base-set (BS)*. Thus for transaction T_i, $\mathcal{BS}_i = \mathcal{RS}_i \cup \mathcal{WS}_i$.

[Özs91] □

Definition 3.3 — *Transaction termination*

A transaction can terminate by *aborting* or *committing*, denoted by a and c respectively. We denote by N_i the termination condition for T_i, where $N_i \in \{a, c\}$.

[Özs91] □

Definition 3.4 — *Conflicting operations*

Two operations $O_i(X)$ and $O_j(X)$ *conflict* if $O_i = w$ or $O_j = w$ and they operate on the same data item X.

[Els94, Özs91] □

In other words, two operations are said to be *conflicting* if they access the same data item and at least one of them is a write.

3.3.2 A model for transaction execution

Having defined the basic transaction concepts, we can continue with the presentation of the model for transaction execution which is used fairly often in the literature [Özs90, Özs91, Bar91, Ber87]:

Definition 3.5 — *Transaction*

A *transaction* T_i is formally defined as a *partial order* $T_i = \{\Sigma_i, \prec_i\}$ where:

1. Σ_i is the domain of T_i and consists of the operations of the transaction and the termination condition, i.e. $\Sigma_i = OS_i \cup \{N_i\}$.
2. $\prec_i$ is an irreflexive and transitive binary relation indicating the execution order of these operations in the transaction; i.e. for any two operations $O_{ij}, O_{ik} \in OS_i$, if $O_{ij} = r(X)$ and $O_{ik} = w(X)$, for any data item X, then either $O_{ij} \prec_i O_{ik}$ or $O_{ik} \prec_i O_{ij}$ or $j = k$.
3. $\forall\, O_{ij} \in OS_i\ ,\ O_{ij} \prec_i N_i$.
4. $a_i \in T_i$ iff $c_i \notin T_i$.

[Özs91, Ber87] □

The first condition defines the domain of the transaction as a set of read and write operations as well as the termination conditions for the transaction, either commit or abort. The second condition specifies the ordering relation between conflicting operations of the transaction. The third condition indicates that the termination condition always follows all the other operations. The last condition states that a transaction must either abort or commit but not both.

Using the above definitions, conflicting transactions can be defined [Tan93]. It is important to know which transactions conflict in order to determine which transactions have to have their order of execution controlled in order to maintain database consistency.

Definition 3.6 — *Conflicting transactions*

A transaction T_j is said to *conflict* with another transaction T_i if any operation of T_j conflicts with any operation of T_i and the operation in T_i precedes the operation in T_j. The conflict relationship is denoted by $T_i \rightsquigarrow T_j$. The transitive closure is denoted by $\overset{*}{\rightsquigarrow}$.

[Meh92c] □

Concurrency control and recovery mechanisms are mainly concerned with the database access commands in a transaction.

Consider the transaction T_1 illustrated in Figure 3.1. According to our formal notation, the specification for the transaction is:

$$\Sigma_1 = \{r(X), r(Y), w(X), w(Y), c\}$$

$$\prec_1 = \{(r(X), w(X)), (r(Y), w(Y)), (r(X), c), (w(X), c), (r(Y), c), (w(Y), c)\}$$

T_1	T_2
read-item(X);	read-item(X)
X:=X-N;	X := X + M
write-item(X);	write-item (X)
read-item(Y);	commit
Y:=Y+N;	
write-item(Y);	
commit	

Figure 3.1: Two sample transactions T_1 and T_2

Transactions submitted by various users may execute concurrently and may access and update the same database items. If this concurrent execution is uncontrolled it may lead to problems such as an inconsistent database. In section 3.5 we discuss the problems that may occur when concurrent transactions do not execute in a controlled manner. In the following section we take a brief look at the different types of transactions which may be found in database systems.

3.4 Transaction Models

In the literature, suggested transaction models abound [Gra93]:

- *Flat transactions* are used in all commercially available database systems, and they are about to be used in operating systems and communication systems. The implementation techniques are well understood and so are the limitations. The "all or nothing" characteristic of flat transactions is both a virtue and a vice. It gives the simplest of failure semantics but in the case of failure, the application programmer can either thread his way back through the application logic by repairing this and reestablishing that or he can rollback the transaction and thereby give up everything done so far.

- *Flat transactions with savepoints* give us the option of having a position inside the transaction to which we could step back in case of failure.

  A savepoint is a place in a transaction where the current state of processing is recorded. A handle is returned to the application program which can be used to refer to that savepoint. Now, if a transaction has to do a rollback, it will rollback to a specified savepoint and will find itself re-instated at that same savepoint.

- *Chained transactions* are a variation of flat transactions. The idea of a chained transaction is that the application program commits what has been done so far, releases all objects no longer needed and waives its right to do a rollback; but at the same time

stays within a transaction. It does not lose database objects acquired during the previous (committed) part of the transaction. The commitment of the first transaction and the beginning of the next are wrapped together into one atomic transaction.

- *Nested transactions* are a generalization of savepoints. Savepoints allow organizing a transaction into a sequence of actions that can be rolled back individually, and nested transactions form a hierarchy of pieces of work. Rather than taking a savepoint after each partial execution, each of the subtransactions becomes a self-contained but dependent action which can be computed or rolled back individually.

  Nested transactions have the following properties:

  1. A nested transaction is a tree of transactions, the sub-trees of which are either nested or flat transactions.
  2. Transactions at leaf level are flat transactions.
  3. The transaction at the root of the tree is called the top level, the others are called subtransactions.
  4. A subtransaction can either commit or rollback; its commit will not take effect unless the parent commits. The subtransaction can only finally commit if the root commits.
  5. The rollback of a transaction anywhere in the tree causes all subtransactions to roll back.

- *Multi-level transactions* are a generalized and more liberal version of nested transactions. They allow for the early commit of a subtransaction and thereby give up the possibility of unilateral backout of the updates. In the case of a subtransaction which should not have committed, it is assumed that there will be a compensating transaction which will reverse the effects of the committed subtransaction.

- *Open nested transactions* are the anarchic version of multi-level transactions. Subtransactions can abort or commit independently of the status of the final outcome of the parent transaction. They constitute unprotected actions.

- *Long lived transactions* are transactions which run for a long time. If one were to use a traditional flat transaction, an abort in the last minute of the transaction would lose work done which perhaps took hours to do. We need to find a way to structure the transaction so that loss in the face of failure is minimal. Some solutions to this problem have been proposed in [Kor88, Nod93, Sal89].

3.5 Why Concurrency Control?

Several problems can occur when concurrent transactions interfere with one another during execution [Els94, Bel92]. Five problems can be identified which may occur when transactions

time	T_1	T_2
t1	read-item(X);	
t2	X:=X-N;	
t3		read-item(X)
t4		X := X + M
t5	write-item(X);	
t6	read-item(Y);	
t7		write-item (X)
t8		commit
t9	Y:=Y+N;	
t10	write-item(Y);	
t11	commit	

Figure 3.2: The lost update problem

execute concurrently. We will elaborate on these problems by means of the two transactions in the example in Figure 3.1.

3.5.1 The lost update problem

The lost update problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect. Suppose the transactions T_1 and T_2 are submitted at approximately the same time, and that their operations are interleaved by the operating system as shown in Figure 3.2. The final value of data item X will then be incorrect, because T_2 reads the value of X before T_1 changes it in the database and the updated value resulting from the execution of T_1 is lost [Els94, Bel92].

3.5.2 The temporary update problem.

The temporary update problem occurs when one transaction updates a database item and then the transaction fails for some reason (see section 3.6). The updated item is accessed by another transaction before it is changed back to its original value. We illustrate this situation in Figure 3.3. This example shows the situation where T_1 updates item X and then fails before completion, so the system must change X back to its original value. Before it can do that, however, transaction T_2 reads the "temporary" value of X. The value of X thus obtained is called *dirty* data, because it has been created by a transaction that has not completed and committed yet; hence this problem is also known as the *dirty read* problem [Els94].

time	T_1	T_2
t1	read-item(X);	
t2	X:=X-N;	
t3	write-item(X);	
t4		read-item(X)
t5		X := X + M
t6		write-item (X)
t7		commit
t8	read-item(Y);	
t9	abort	

Figure 3.3: The temporary update problem

3.5.3 The incorrect summary problem

If one transaction is calculating an aggregate summary function on a number of records while another transaction is updating some of the values, the aggregate may read some values before they have been updated and others after they have been updated [Els94]. This type of situation is illustrated by Figure 3.4. This example shows a situation where transaction T_3 reads data item X after N is subtracted, and reads Y before N is added, so a wrong summary is the result (which will be off by N).

3.5.4 Violation of integrity constraints

This problem can occur if two transactions are allowed to execute concurrently without being synchronized [Bel92]. Consider a hospital database containing two relations:

SCHEDULE (Surgeon_name, Operation, Date).
SURGEON (Surgeon_name, Operation).

SCHEDULE specifies which surgeon is scheduled to perform a particular operation on a certain date. The SURGEON relation records the qualifications by operation for each surgeon. An important integrity constraint for this database is that surgeons must be qualified to perform operations for which they are scheduled. The initial state of the database is shown in Figures 3.5 and 3.6.

Suppose there are two transactions T_3 and T_4 which concurrently access the database. The transaction details are given in Figure 3.7.

Transaction T_3 changes the operation scheduled on 04.04.95 from a tonsillectomy to an appendectomy. It does integrity checking before making the change. Meanwhile, it is discovered independently that Mary will not be able to do the operation as she is otherwise engaged so a transaction T_4 then changes the surgeon to Tom after checking that Tom is able to carry out the operation currently scheduled for 04.04.95, namely a tonsillectomy. The

time	T_1	T_3
t1		sum := 0
t2		read-item(A);
t3		sum := sum+A;
t4		• • •
t5	read-item(X);	
t6	X:=X-N;	
t7	write-item(X)	
t8		read-item(X)
t9		sum := sum + X;
t10		read-item(Y);
t11		sum := sum + Y;
t12	read-item(Y);	
t13	Y:=Y+N;	
t14	write-item(Y);	
t15	commit	
t16		commit

Figure 3.4: The incorrect summary problem

Surgeon_name	Operation	Date
Mary Jones	Tonsillectomy	04.04.95
•	•	•
•	•	•
•	•	•

Figure 3.5: Initial state of SCHEDULE table

Surgeon_name	Operation
Tom Bones	Tonsillectomy
Mary Jones	Tonsillectomy
Mary Jones	Appendectomy
•	•
•	•
•	•

Figure 3.6: Initial state of SURGEON table

time	T_3	T_4
t1	read-item(Surgeon_name, Operation, Date) in SCHEDULE where date="04.04.95"	
t2		read-item(Surgeon_name, Operation, Date) in SCHEDULE where date="04.04.95"
t3	read-item(Surgeon_name) in SURGEON where SCHEDULE.Surgeon_name = SURGEON.Surgeon_name and where SURGEON.Operation = "Appendectomy"	
t4		read-item(Surgeon_name) in SURGEON where SCHEDULE.Surgeon_name = "Tom Bones" and SURGEON.Operation = SCHEDULE.Operation
t5	If not found, then ABORT T_3 (See note 1 below) ELSE, SCHEDULE.Operation = "Appendectomy"	
t6		If not found, then ABORT T_4 (See note 2 below) ELSE, SCHEDULE.Surgeon_name = "Tom Bones"
t7	COMMIT(T_3)	
t8		COMMIT(T_4)
	1: Indicates switch of operations not possible because surgeon scheduled on 04.04.95 is not qualified to perform the new operation.	
		2: Indicates switch of surgeons not possible because new surgeon is not qualified to perform operation scheduled.

Figure 3.7: Operations of transactions T_3 and T_4

effect of these two operations is to produce an inconsistent database. Neither transaction is aware of the other as they are updating different data.

3.5.5 The unrepeatable read

This occurs where a transaction T_i reads an item twice, and the item is changed between the two reads by another transaction. Hence, T_i receives different values for its two reads of the same item [Els94].

3.6 Why Recovery?

Whenever a transaction is submitted to a DBMS for execution, the system is responsible for making sure that either:

- all the operations in the transaction are completed successfully and their effect is recorded permanently in the database, or
- the transaction has no effect whatsoever on the database or on any other transactions. The DBMS must not permit some operations of a transaction T_i to be applied to the database while other operations of T_i are not. This may happen if a transaction fails after executing some of its operations but before executing all of them [Els94].

There are various reasons why a transaction can fail in the middle of execution [Els94]:

1. *A computer failure (system crash)*: This could be caused by either a hardware or software failure.
2. *A transaction or system error*: Some operation in the transaction may cause it to fail, such as integer failure or division by zero. On the other hand the user may abort the transaction physically.
3. *Local errors* or *exception conditions* detected by the transaction: During transaction execution, the transaction may need to be cancelled. This could happen if the required record in the database could not be found, for example.
4. *Concurrency control enforcement*: The transaction may have to be aborted because of a deadlock situation or because the concurrency control method decides to abort it.
5. *Disk failure*: Some disk blocks may lose their data because of a read/write problem. This could also be caused by, for example, virus activity.
6. *Physical problems* and *catastrophes*: This refers to power failures, theft, sabotage etc.

7. *Timeouts*: If a transaction is present in a system for longer than a specified time, the system could abort the transaction because of a timeout failure. The timeout value is system dependent.

Failures of types 1 to 4 happen more frequently than the others. For failures 1 to 4 the system must have sufficient information about the transaction to recover from the failure. It is very difficult to recover from failures of types 5 and 6.

Recovery techniques for centralized database systems can be divided into seven different categories [Ver78]:

1. *Salvation program*: A salvation program is run after a crash to restore the system to a valid state. It uses no recovery data. It is used after a crash if other recovery techniques fail or are not used, or if no crash resistance is provided. The program scans the database after a crash to asses the damage and to restore the database to some valid state. It rescues the information that is still recognizable.

2. *Incremental dumping*: This involves the copying of updated files onto archival storage after a job has finished or at regular intervals. It creates checkpoints for updated files. Backup copies of files can then be restored after a crash.

3. *Audit trail*: An audit trail records sequences of actions on files. It can be used to restore files to their states prior to a crash or to roll back a particular transaction.

4. *Differential files*: A file can consist of two parts: the main file which is unchanged, and the differential file which records all the alterations requested for the main file. The main files are regularly merged with the differential files, thereby emptying the differential files. The differential file also helps implement crash resistance.

5. *Backup/current version*: The files containing the present values of existing files form the current version of the database. Files containing previous values form a consistent backup version of the database. Backup versions can be used to restore files to previous values.

6. *Multiple copies*: More than one copy of each file is held. The different copies are identical except during update. A "lock bit" can be used to protect a file during updating, while its state is inconsistent. If there is an odd number of files, comparison can be done to select a consistent version. This technique provides crash resistance and may also be used to detect faults if different copies are kept on different devices. The difference between multiple copies and backup/current version is that all multiple copies are active at any one time while with backup/current version there is only one active copy.

7. *Careful replacement*: The principle of careful replacement is that it avoids updating any part of the database in place. Altered parts are put in a copy of the original; the

original is only deleted after the alteration is completed and has been certified. In this method two copies exist only during update. This technique also provides crash resistance because the original will always be available in case a crash occurs during update.

A full discussion of these techniques can be found in [Ver78]. Multidatabase recovery concepts have been discussed in greater detail in Chapter 6.

3.7 Transaction Schedules

When transactions are executing concurrently in an interleaved fashion, the *order* of execution of operations from the various transactions forms what is known as a transaction *schedule*. We shall now define the concept of a schedule.

3.7.1 Schedules (histories) of transactions

Definition 3.7 — *Schedule*

Given a DBMS with a set of transactions $\mathcal{T}$ a schedule (S) is a partial order $S = (\Sigma, \prec)$ where:

1. $\Sigma = \bigcup_j \Sigma_j$ where Σ_j is the domain of $T_j \in \mathcal{T}$.
2. $\prec_S \supseteq \bigcup_j \prec_j$ where $\prec_j$ is the ordering relation for transaction T_j at the DBMS.
3. for any two conflicting operations $p, q \in S$, either $p \prec_S q$ or $q \prec_S p$.

[Els94, Bar90] □

A *schedule* S of n transactions $\mathcal{T} = \{T_1, T_2, ... T_n\}$ is an ordering of the operations of the transactions subject to the constraint that, for each transaction T_i that participates in S, the operations of T_i in S must appear in the same order in which they occur in T_i. Note that operations from transaction T_j can be interleaved with the operations of T_i in S

For the purpose of concurrency control and recovery, we are interested in the *read-item* and *write-item* operations of the transaction as well as the *commit* and *abort* operations. We will use the notation r_i, w_i, c_i and a_i for the operations read-item, write-item, commit and abort of transaction T_i respectively. The subscript corresponds to the specific transaction under discussion.

Definition 3.8 — *Complete schedule*

A *complete schedule* S_T^c [Özs91] defined over a set of transactions $\mathcal{T} = \{T_1, T_2, ...T_n\}$ is a partial order $S_T^c = \{\Sigma_T, \prec_T\}$ where:

1. $\Sigma_T = \bigcup_{i=1}^{n} \Sigma_i$.

2. $\prec_T \supseteq \bigcup_{i=1}^{n} \prec_i$.
3. For any two conflicting operations $O_{ij}, O_{kl} \in \Sigma_T$, either $O_{ij} \prec_T O_{kl}$, or $O_{ij} \prec_T O_{kl}$.
4. $a_i \in T_i$ iff $c_i \notin T_i$.

□

Informally: a schedule S of n transactions $\mathcal{T} = \{T_1, T_2, ... T_n\}$ is said to be a *complete schedule* if the following conditions hold:

1. The operations in S are exactly those operations in $T_1, T_2, ... T_n$, including a commit or abort operation as the last operation for each transaction in the schedule.
2. For any pair of operations from the same transaction T_i, their order of appearance in S is the same as their order of appearance in T_i.
3. For any two conflicting operations, one of the two must occur before the other in the schedule.
4. Any transaction in S must either commit or abort but not both.

Example 3.1 : Applying the transaction model to transactions T_1 and T_2

Consider once again transactions the set of transactions $\mathcal{T} = \{T_1 , T_2\}$ in Figure 3.1. We can express these transactions formally in terms of the model outlined above:

$$\Sigma_1 = \{r_1(X), r_1(Y), w_1(X), w_1(Y), c_1\}$$

$$\Sigma_2 = \{r_2(X), w_2(X), c_2\}$$

$$\prec_T = \{(r_1(X), w_1(X)), (r_1(Y), w_1(Y)), (r_1(X), c_1),$$
$$(w_1(X), c_1), (r_1(Y), c_1), (w_1(Y), c_1),$$
$$(r_2(X), w_2(X)), (r_2(X), c_2), (w_2(X), c_2),$$
$$(w_1(X), r_2(X)), (w_2(X), r_1(X)), (c_1, c_2), (c_2, c_1)\}$$

A possible complete schedule for transactions T_1 and T_2 can be specified as:

$$S_T^c = \{r_1(X), w_1(X), r_2(X), w_2(X), r_1(Y), w_1(Y), c_1, c_2\}$$

We will use the notation of S_x where S refers to a particular schedule of the operations in T_1 and T_2 and x denotes a particular ordering of those operations that will be used.

Using this notation, a possible schedule for transactions T_1 and T_2 could be:

S_a : $r_2(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; c_2; $w_1(Y)$; c_1;

Similarly, a schedule for T_1 and T_2 in Figure 3.1 can be written as follows if it is assumed that T_1 aborts after its *read-item(Y)* operation:

S_b : $r_1(X)$; $w_1(X)$; $r_2(X)$; $w_2(X)$; c_2; $r_1(Y)$; a_1;

◇

It is very difficult to encounter complete schedules in a transaction processing environment, because new transactions are continually being introduced into the system [Els94]. Hence, the concept of a *committed projection* $C(S_T^c)$ of a schedule S is introduced. This includes only the operations in S that belong to committed transactions — that is, transactions T_i whose commit operation c_i is in S.

Definition 3.9 — *Projection of a schedule*

A projection of a schedule S on a set of transactions T' is:
$S^{T'} = (T', \prec_{S^{T'}})$ where $\prec_{S^{T'}} \subseteq \prec_S$ such that for all O_i, O_j operations in T', $O_i \prec_S O_j$ iff $O_i \prec_{S^{T'}} O_j$.

[Meh92c]. □

A projection of a schedule S on a set of transactions T' denoted by $S^{T'}$ is a schedule obtained from S by deleting all operations that do not belong to transactions in T'

Definition 3.10 — *Committed projection*

A *committed projection* $C(S_T^c)$ of S_T^c can de defined as: Given a partial order $S_T^c = \{\Sigma, \prec\}$ then $C(S_T^c) = \{\Sigma', \prec'\}$ where:

1. $\Sigma' \subseteq \Sigma$.
2. $\forall$ elements $e_i \in \Sigma'$, $e_1 \prec' e_2$ iff $e_1 \prec e_2$.
3. $\forall\ e_i \in \Sigma'$, if $\exists e_j \in \Sigma$ and $e_j \prec' e_i$, then $e_j \in \Sigma'$.
4. For each T_i in $C(S_T^c)$, there exists some pair (O_i, c_i) or (O_i, a_i) in $\prec'$ where $O_i \in \{r, w\}$.

[Ber87] □

To obtain the complete schedule, simply delete all operations that belong to transactions that have not yet committed in S.

All schedules from here onwards will be considered to be committed projections of complete schedules. In the next section the question of why schedules are so important when considering concurrency and recovery in database systems is discussed.

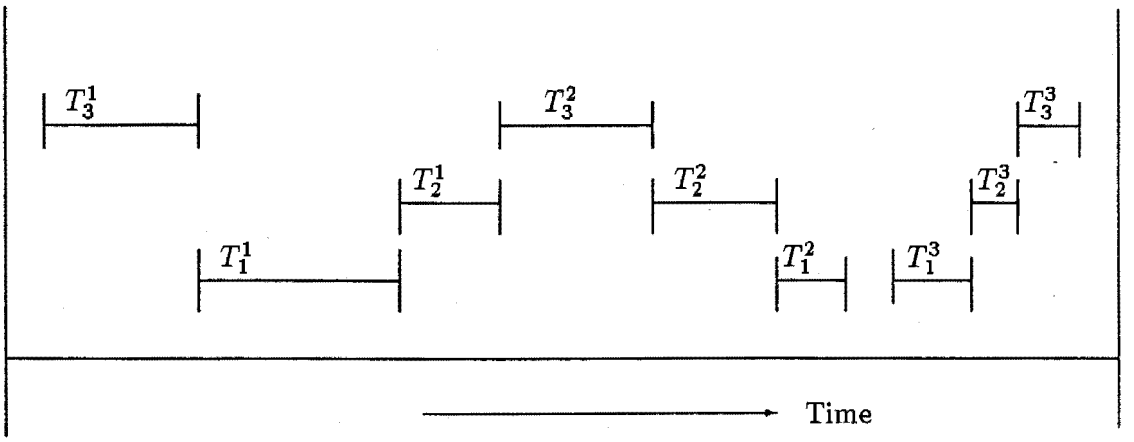


Figure 3.8: End to end transaction execution

[Bel92, p.167]

3.7.2 Serializability of schedules

Serializability theory attempts to determine which schedules are correct and which are not in order to develop techniques that allow only correct schedules [Els94]. This section will define the concepts of serializability theory.

To understand serializability, one must first define a serial schedule because a serial schedule causes transactions to execute consecutively. This serial execution of a set of transactions will always maintain the consistency of a database [Özs91].

Definition 3.11 — *Serial schedule*

A database schedule S for a set of n transactions $\mathcal{T} = \{ T_1,...,T_r \}$ is serial iff
$(\exists O_i \in T_i, \exists O_j \in T_j$ such that $O_i \prec_S O_j) \models (\forall O_r \in T_i, \forall O_u \in T_j, O_r \prec_S O_u)$.

[Bar90] □

A *serial schedule* is one in which all the reads and writes of each transaction are grouped together so that the transactions are run one after the other, otherwise it is a *non-serial schedule.* Serial execution of transactions is illustrated in Figure 3.8.

A schedule S is said to be *serializable* if all the reads and writes of each transaction can be reordered in such a way that when they are grouped together as in a serial schedule, the net effect of executing this reorganized schedule is the same as that of the original schedule S. This reorganized schedule is called the *equivalent serial schedule.* A serializable schedule will therefore be equivalent to, and have the same effect on the database, as some serial schedule.

Definition 3.12 — *Conflict equivalence of schedules ($\equiv$)*

Two schedules S_a and S_b are *conflict equivalent* if:

1. They are defined over the same set of transactions with identical operations, and
2. they order conflicting operations of non-aborted transactions in the same way; that is, for any conflicting operations O_i and O_j belonging to transactions T_i and T_j (respectively) where a_i, $a_j \notin S$, if $O_i \prec_{S_a} O_j$ then $O_i \prec_{S_b} O_j$.

[Bar90, Ber87] □

Recall that two operations in a schedule are said to conflict if they belong to different transactions, if they access the same data item, and if one of the two operations is a write-item operation. If two conflicting operations are applied in different orders in two schedules, the effect of the schedules can be different on either the transactions or the database, and hence the schedules are not conflict equivalent [Els94].

Definition 3.13 — *Serializable / Conflict serializable*

Using the notion of conflict equivalence, we define a schedule to be *conflict serializable* (also referred to as simply serializable) if it is equivalent to some serial schedule S' .

[Özs91] □

A serializable schedule is *not* the same as a serial schedule. The objective of the concurrency control algorithm is to produce correct schedules so that the transactions are scheduled in such a way that they transform the database from one consistent state to another consistent state and do not interfere with one another.

Serializability is taken as proof of correctness. Thus, if the concurrency control algorithm generates serializable schedules, then these schedules are guaranteed to be correct. Deciding whether a schedule is equivalent to a serial schedule is difficult. Intuitively, we can say that two schedules are equivalent if their effect on the database is the same. Thus each read on data item X in both schedules sees the same value for X and the final write operation on each data item will be the same in both schedules. In terms of schedule equivalence, it is the ordering of conflicting operations which must be the same in both schedules [Bel92].

In a serializable schedule, we can reorder the non-conflicting operations in S until we form the equivalent serial schedule S'. To illustrate this, consider the transactions T_1 and T_2 defined in Figure 3.1. In Figures 3.9 to 3.12, we can see four possible schedules for these two transactions.

time	T_1	T_2	Value of X in the database
t1	read-item(X);		90
t2	X:=X-N;		
t3	write-item(X);		87
t4	read-item(Y);		
t5	Y:=Y+N;		
t6	write-item(Y);		
t7	commit		
t8		read-item(X)	87
t9		X := X+M	
t10		write-item(X)	89
t11		commit	

Figure 3.9: Schedule (a) involving transactions T_1 and T_2

time	T_1	T_2	Value of X in the database
t1		read-item(X)	90
t2		X := X+M	
t3		write-item(X)	92
t4	read-item(X);		92
t5		commit	
t6	X:=X-N;		
t7	write-item(X);		89
t8	read-item(Y);		
t9	Y:=Y+N;		
t10	write-item(Y);		
t11	commit		

Figure 3.10: Schedule (b) involving transactions T_1 and T_2

time	T_1	T_2	Value of X in the database
t1	read-item(X);		90
t2	X:=X-N;		
t3		read-item(X)	90
t4		X := X+M	
t5	write-item(X);		87
t6	read-item(Y);		
t7		write-item(X)	92
t8		commit	
t9	Y:=Y+N;		
t10	write-item(Y);		
t11	commit		

Figure 3.11: Schedule (c) involving transactions T_1 and T_2

time	T_1	T_2	Value of X in the database
t1	read-item(X);		90
t2	X:=X-N;		
t3	write-item(X);		87
t4		read-item(X)	87
t5		X := X+M	
t6		write-item(X)	89
t7		commit	
t8	read-item(Y);		
t9	Y:=Y+N;		
t10	write-item(Y);		
t11	commit		

Figure 3.12: Schedule (d) involving transactions T_1 and T_2

According to the definition of conflict serializability, schedule (d) in Figure 3.12 is equivalent to the serial schedule (a) in Figure 3.9. In both these schedules, the $r_2(X)$ of T_2 reads the value of X written by T_1, while the other read-item operations read the database values from the initial database state. In addition, T_1 is the last transaction to write item Y, and T_2 is the last transaction to write X in both schedules. Because schedule (d) is equivalent to serial schedule (a), (d) is a serializable schedule. Schedule (c) is not equivalent to either of the serial schedules (a) or (b) so (c) is not serializable. A simple algorithm for testing the conflict serializability of a schedule is outlined in [Els94].

The concept of serializability of schedules has been examined and we can now proceed to discussing whether schedules are recoverable or not.

3.7.3 Recoverable schedules

In order to ensure correctness in the presence of failures, the scheduler must produce executions that are not only serializable but also recoverable. For some schedules it is easy to recover from transaction failures, whereas for others the recovery process is quite involved. Hence it is important to characterize the types of schedules for which recovery is possible, as well as those for which recovery is quite simple. First of all, it is desirable to ensure that once a transaction has committed, it should never be necessary to roll back a transaction T_i. The schedules that meet this criterion are called *recoverable schedules.*

A transaction T_i is said to *read from* transaction T_j in a schedule if some item X is first written by T_j and later read by T_i.

Definition 3.14 — *T_i reads from T_j*

We say transaction T_i *reads data item X from T_j* in S if

1. $w_j(X) \prec r_i(X)$
2. $a_j \not\prec r_i(X)$ [2], and
3. if there is some $w_k(X)$ such that $w_j(X) \prec w_k(X) \prec r_i(X)$, then $a_k \prec r_i(X)$.

[Ber87] □

In the schedule S, T_j should not have aborted before T_i reads item X, and there should be no transactions that write X after T_j writes it and before T_i reads it (unless those transactions, if any, have aborted before T_i reads X). It is also possible for a transaction to read from itself.

[2] $p \not\prec q$ denotes that operation p does not precede q in the partial order

Definition 3.15 — *Recoverability*

A schedule S is called *recoverable* if:

1. Whenever T_i reads from T_j $(i \neq j)$ in S, and
2. $c_i \in S$, then $c_j \prec c_i$.

[Ber87] □

A schedule is said to be *recoverable* if no transaction T_i in S commits until all transactions T_j that have written an item that T_i reads have committed.
These concepts are best illustrated by an example.

Example 3.2 : Recoverability of schedules

Consider the following schedules for transactions T_1 and T_2:

$$S_c = r_1(X), w_1(X), r_2(X), r_1(y), w_2(X), c_2, a_1;$$

$$S_d = r_1(X), w_1(X), r_2(X), r_1(y), w_2(X), w_1(y), c_1, c_2;$$

S_c is not recoverable because T_2 reads item X from T_1, and then T_2 commits before T_1 commits. If T_1 aborts after the c_2 operation in S_c, then the value of X that T_2 read is no longer valid and T_2 must be aborted after it has already committed, leading to a schedule that is not recoverable. For the schedule to be recoverable, the c_2 operation in S_c must be postponed until after T_1 commits, as shown in S_d.

In a recoverable schedule, no committed transaction ever needs to be rolled back but a phenomenon known as *cascading rollback* (or cascading abort) can still occur. This happens when an uncommitted transaction has to be rolled back because it read an item from a transaction that failed. This is illustrated in schedule S_e, where transaction T_2 has to be rolled back because it read item X from T_1, and T_1 then aborted.

$$S_e = r_1(X), w_1(X), r_2(X), r_1(y), w_2(X), w_1(y), a_1;$$

Cascading rollback is quite time consuming so we need to characterize schedules where the problem does not present. ◇

Definition 3.16 — *Avoids cascading rollbacks*

A schedule S *avoids cascading rollbacks* if:

Whenever transaction T_i reads X from T_j $(i \neq j)$, then $c_j \prec r_i(X)$.

[Ber87]

□

A schedule is said to avoid cascading rollbacks if every transaction in the schedule only reads items that were written by committed transactions.

To satisfy this criterion, the $r_2(X)$ command in schedule S_e (shown above) must be postponed until after T_1 has committed (or aborted), thus delaying T_2 but ensuring no cascading rollback if T_1 aborts.

Finally, there is a third, more restrictive type of schedule, called a *strict* schedule, in which transactions can neither read nor write an item X until the last transaction that wrote X has committed (or aborted). Strict schedules simplify the process of recovering write operations to a matter of restoring the before image of a data item X, which is the value that X had prior to the aborted write operation.

Definition 3.17 — *Strict*

A schedule S is *strict* if:

Whenever $w_j(X) \prec O_i(X)$ $i \neq j$,

1. either $a_j \prec O_i(X)$, or
2. $c_j \prec O_i(X)$ where $O_i(X)$ is $r_i(X)$ or $w_i(X)$.

[Ber87] □

Once again, this concept is best illustrated by an example:

Example 3.3 : A non-strict schedule

Consider the schedule:

$S_f = w_1(X,5), w_2(X,8), a_1;$

Suppose the value of X was 9 originally. When transaction T_1 aborts, the value of X will be restored to 9 (which is not correct because T_2 already changed the value to 8). This happens because T_2 was allowed to write X even though the previous transaction that wrote to X had not yet committed. A strict schedule does not have this problem. ◇

Bernstein [Ber87] proves that *recoverability* (RC), *avoiding cascading aborts* (ACA) and *strictness* (ST) are increasingly restrictive properties. i.e. $ST \subset ACA \subset RC$

We can now develop a classification of schedules based on conflicts between various transactions in a schedule S. This helps us to identify the requirements of S when defining still further restrictions on schedules [Meh92c].

- **ROW**: For all pairs of transactions T_i, T_k in S, if T_i reads a data item X that is later written by T_k, then T_k does not commit before T_i either commits or aborts.
- **AROW**: For all pairs of transactions T_i, T_k in S, if T_i reads a data item X that is later written by T_k, then T_k does not write on X before T_i either commits or aborts.
- **WOR**: For all pairs of transactions T_i, T_k in S, if T_i writes a data item X that is later read by T_k, then T_k does not commit before T_i either commits or aborts.
- **AWOR**: For all pairs of transaction T_i, T_k in S, if T_i writes data item X that is later read by T_k, then T_k does not read X before T_i either commits or aborts.
- **WOW**: For all pairs of transactions T_i, T_k in S, if T_i writes on a data item X that is later written by T_k, then T_k does not commit before T_i either commits or aborts.
- **AWOW**: For all pairs of transactions T_i, T_k in S, if T_i writes on data item X that is later written by T_k, then T_k does not write on X before T_i either commits or aborts.

Certain combinations of these classes of schedules can now be identified [Meh92c]:

Definition 3.18 — *Rigorous schedule*

A schedule is rigorous if it is AROW and AWOR and AWOW.

[Bre91a] □

Definition 3.19 — *Strongly recoverable schedule*

A schedule is strongly recoverable if it is ROW and WOR and WOW.

[Bre91a] □

Definition 3.20 — *Semi-rigorous schedule*

A schedule is semi-rigorous if it is ROW and AWOR and WOW.

[Meh92c] □

The concepts in the above definitions have been used in later chapters when the principles outlined here are applied to multidatabases. Concurrency control in databases is concerned with preventing interference or conflict between concurrently executing transactions. The techniques for concurrency control are discussed in section 3.8.

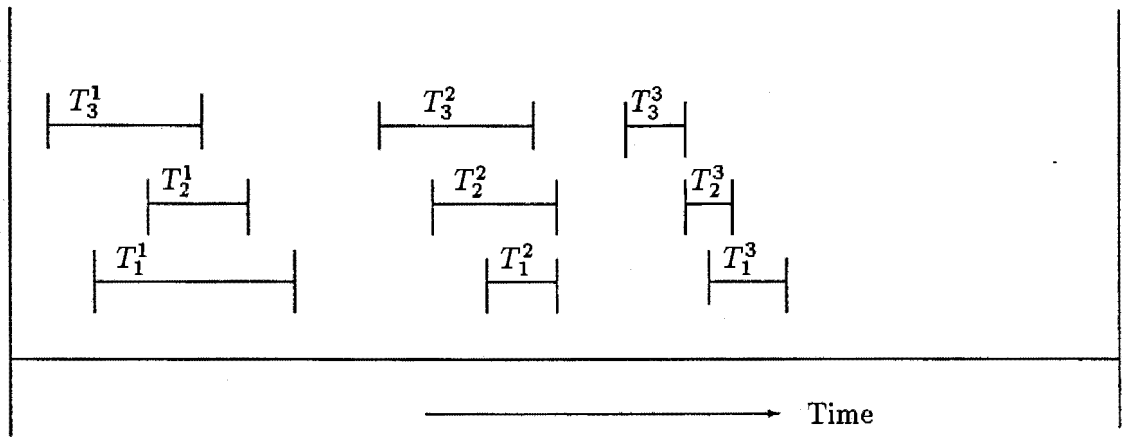


Figure 3.13: Concurrent execution of transactions

[Bel92, p.157]

3.8 Concurrency Control Techniques in Centralized Databases

Figure 3.13 illustrates concurrently executing transactions.

Example 3.4 : Problems with concurrently executing transactions

To illustrate the problems which can be caused by concurrently executing transactions, consider the transaction schedules shown in Figures 3.9 to 3.12. Assume that the starting values for X=90, Y=90 and that N=3, M=2. After executing transactions T_1 and T_2, we would expect data values to be X=89 and Y=93. If schedules (a) or (b) are executed, the values will be correct. If schedule (c) is considered, the results X=92 and Y=93 are obtained, in which X is erroneous whereas (d) gives the correct results. Schedule (c) gives the wrong results because of the lost update problem. Transaction T_2 reads the value of X before it is changed by transaction T_1, so only the effect of transaction T_2 is seen in the database. ◇

Numerous concurrency control methods have been proposed in the literature. Depending on the behaviour of the concurrency control mechanism, the mechanisms can be classified as *schedulers* or *certifiers*. In addition, schedulers can be classified as *conservative* or *aggressive*. Conservative schedulers tend to delay operations that introduce conflicts. Aggressive schedulers tend to reject such operations, and cause the transaction that issued them to abort. While schedulers determine whether an operation will cause a conflict before they execute it, certifiers execute all operations immediately. Certifiers test completed but not yet committed transactions to determine whether their execution was serializable. If a

transaction is found to violate serializability, it is aborted. Otherwise, it is permitted to commit [Geo90].

There are five basic concurrency control techniques which allow transactions to execute safely in parallel subject to certain constraints:

1. Timestamp methods
2. Locking methods
3. Optimistic methods
4. Serialization graph methods
5. Value date methods.

Locking, timestamping and value date are schedulers because they delay transactions in case they conflict with other transactions at some time in the future. Optimistic and serialization graph methods allow transactions to proceed unsynchronized and only check for conflicts just before a transaction commits and are thus certifiers.

3.8.1 Timestamp methods

3.8.1.1 Basic timestamping algorithm

In basic timestamping, each transaction T_i is given a unique timestamp $ts(T_i)$ at its initiation. These timestamp values are derived from a totally ordered domain. A timestamp is not a transaction identifier. With the timestamping algorithm it is simple to order transactions' operations according to their timestamps.

Definition 3.21 — *Ageing transactions*

> Given two conflicting operations O_{ij} and O_{kl} on a database item X belonging respectively to transactions T_i and T_k, $O_{ij}(X)$ is executed before $O_{kl}(X)$ if and only if $ts(T_i) < ts(T_k)$. In this case T_i is said to be the *older transaction* and T_k is said to be the *younger transaction.*
>
> [Els94] □

A scheduler checks each new operation against conflicting operations that have already been scheduled. If the new operation belongs to a younger transaction than all the conflicting ones that have already been scheduled, the operation is accepted; otherwise it is rejected, causing the entire transaction to restart with a new timestamp.

There can be no deadlock (see section 3.8.2.1) because transactions do not wait for each other. If two transactions conflict, one of them is simply rolled back and restarted. The fundamental goal is to order transactions globally in such a way that older transactions,

i.e. transactions with smaller timestamps, get priority in the event of a conflict. If a transaction attempts to read or write a data item, then the read or write will only be allowed to proceed if the last update was carried out by an older transaction; otherwise the requesting transaction will be rolled back and restarted with a new timestamp.

However, the comparison between timestamps can only be done when all the operations to be scheduled have been received by the scheduler. If they arrive one at a time (the most probable scenario), it is necessary to check if an operation has arrived out of sequence. To facilitate this check, each data item X is assigned two timestamps: a *read timestamp [rts(X)]*, which is the largest of the timestamps of the transactions that have read X, and a *write timestamp [wts(X)]*, which is the largest of the timestamps of the transactions that have written (updated) X. It is now sufficient to compare the timestamp of an operation with the read and write timestamps of the data item that it wants to access to determine if any transaction with a larger timestamp has already accessed the data item. To express it formally:

Definition 3.22 — *Basic timestamp ordering*

1. If transaction T_i issues a $w_i(X)$ operation:
   (a) If $rts(X) > ts(T_i)$, or if $wts(X) > ts(T_i)$, then abort and roll back T_i and then reject the operation. This should be done because some transaction with a timestamp greater than $ts(T_i)$ — and hence after T_i in the timestamp ordering — has already read or written the value of data item X before T_i had a chance to write X, thus violating timestamp ordering.
   (b) If the condition in part (a) does not occur, then execute the $w_i(X)$ operation of T_i and set $wts(X)$ to $ts(T_i)$.
2. If transaction T_i issues a $r_i(X)$ operation:
   (a) If $wts(X) > ts(T_i)$, then abort and roll back T_i and reject the operation. This should be done because some transaction with a timestamp greater than $ts(T_i)$ — and hence after T_i in the timestamp ordering — has already read or written the value of X before T_i had a chance to read X, thus violating timestamp ordering.
   (b) If $wts(X) \leq ts(T_i)$, then execute the $r_i(X)$ operation of T_i and set the $rts(X)$ to the larger of $ts(T_i)$ and the current $rts(X)$.

[Els94] □

Hence, the basic timestamp ordering algorithm checks whether two conflicting transactions occur in the incorrect order, and rejects the later of the two operations by aborting the transaction that issued it. The schedules produced by this algorithm are always serializable, which are equivalent to the serial schedule defined by the timestamps of successfully

committed transactions [Bel92, Els94]. Unfortunately this algorithm does not produce recoverable, cascadeless or strict schedules. A variation of basic timestamp ordering called *strict timestamp ordering* ensures schedules that are both strict and conflict serializable. This algorithm works as follows:

Definition 3.23 — *Strict timestamp ordering*

> If a transaction T_i issues a $r_i(X)$ or a $w_i(X)$ such that $ts(X) > wts(X)$ then the operation is delayed until the transaction T_j that wrote the value of X (hence $ts(T_j) = wts(X)$) has committed or aborted.
>
> [Els94] □

To implement the algorithm, it is necessary to simulate the locking of an item X that has been written by transaction T_j until T_j is either committed or aborted. This algorithm cannot cause deadlock because T_i only waits for T_j if $ts(T_i) > ts(T_j)$ [Els94].

3.8.1.2 Conservative timestamp ordering rule

In the basic algorithm operations are never delayed, but instead transactions are simply restarted. Although this causes a deadlock free system, it causes numerous restarts and adversely affects performance. The conservative algorithm attempts to lower overhead by reducing the number of transaction restarts. The conservative timestamp ordering rule algorithm will delay each operation until there is an assurance that no operation with a smaller timestamp can arrive at the scheduler which reduces the problem of frequent restarts but on the other hand introduces a deadlock possibility.

The basic technique here is based on the following: the operations of each transaction are buffered until an ordering can be established so that rejections are not possible, and they are executed in that order.

The timestamp algorithm then actually executes transactions serially at each site. This is very restrictive [Özs91].

3.8.1.3 Multiversion timestamp ordering rule

Multiversion Timestamp Ordering *(TO)* is an attempt at eliminating the restart overhead cost of transactions. In multiversion *TO*, the updates do not modify the database; each write operation creates a new copy of that data item. Each version is marked by the timestamp of the transaction that created it. This algorithm trades storage space for time. It then processes each transaction serially in timestamp order. The existence of versions is transparent to the user because the scheduler will ensure that the user reads the latest value for a particular data item. The scheduler simply assigns a timestamp to each transaction which is also used to keep track of the timestamps of each version.

The operations are processed by the scheduler as follows:

Definition 3.24 — *Multiversion timestamp rule*

1. A $r_i(X)$ is translated to a read on one version of X. This is done by finding a version of X (say X_v) such that $ts(X_v)$ is the largest timestamp less than $ts(T_i)$. $r_i(X_v)$ is then sent to the data processor.
2. A $w_i(X)$ is translated to $w_i(X_w)$ so that $ts(X_w) = ts(T_i)$ and sent to the data processor if and only if no other transaction with a timestamp greater than $ts(T_i)$ has read the value of a version of X (say X_r) such that $ts(X_r) > ts(X_w)$. In other words, if the scheduler has already processed a $r_j(X_r)$ such that $ts(T_i) < ts(X_r) < ts(T_j)$ then $w_i(X)$ is rejected.

[Els94] □

A scheduler that processes the read and write operations of transactions according to the rules above is guaranteed to generate serializable schedules [Özs91]. To save space, the DBMS may purge some of the versions from time to time. This will be done when the DBMS is sure that no transaction will require the purged version again.

3.8.2 Locking methods

This is the most widely used approach to handling concurrency control in DBMSs. A transaction must claim a *(shared) read lock (rl)* or *(exclusive) write lock (wl)* lock on a data item prior to the execution of the corresponding read or write operation on that data item. Since read operations cannot conflict, it is permissible for more than one transaction to hold read locks for a data item. A write lock, however, gives a transaction exclusive access to a data item. As long as a transaction holds a write lock on a data item, no other transaction can either read or write that data item.

A typical schedule for the transactions in Figure 3.1 would be:

$$S = \{wl_1(X), r_1(X), w_1(X), rl_1(X), wl_2(X), wl_1(Y), r_1(Y), r_2(Y), \\ w_1(Y), w_2(X), c_1, c_2\}$$

The most common locking protocol is known as *two-phase locking* ($2PL$). The transactions which obey this protocol operate in two distinct phases: a growing phase during which the transaction acquires all the locks, and a shrinking phase during which it releases those locks (denoted by $rell$).

A typical schedule for the transactions in Figure 3.1 using the $2PL$ protocol would be:

$$S = \{wl_1(X), wl_1(Y), r_1(X), w_1(X), rell_1(X), wl_2(X), r_1(Y), r_2(X), \\ w_1(Y), w_2(X), rell_2(X), rell_1(Y), c_1, c_2\}$$

There are a number of variations of $2PL$. The technique described above is known as *basic 2PL*. A variation known as *conservative 2PL* requires a transaction to lock all the items it accesses before the transaction begins execution, by predeclaring its read set and write set. Conservative 2PL is a deadlock free protocol. In practise, the most popular version of 2PL is *strict 2PL*, which guarantees strict schedules. In this variation, the transaction does not release any of its locks until after it commits or aborts. Hence no other transaction can access a data item until after the other transaction has committed or aborted which leads to a strict schedule. Strict 2PL (also sometimes called *rigorous 2PL* [Bre95]) is not deadlock-free unless it is combined with conservative 2PL. Of course, the use of locks leads to the problems of deadlock. How to deal with this is discussed next.

3.8.2.1 Deadlock

Deadlock occurs when each of two transactions is waiting for the other to release the lock on an item. Informally, a deadlock is a set of requests that can never be granted by the concurrency control mechanism.

Example 3.5 : Deadlock

For example, consider two transactions T_i and T_j that hold locks on two entities X and Y (i.e. $wl_i(X)$ and $wl_j(Y)$). Suppose now that T_i issues a $rl_i(Y)$ or a $wl_i(Y)$. Since Y is currently locked by T_j, T_i will have to wait until transaction T_j releases its lock on Y. However, if during this waiting period, T_j now requests a lock (read or write) on X, there will be a deadlock. This is because T_i will be blocked waiting for T_j to release its lock on Y while T_j is blocked waiting for T_i to release its lock on X. In this case the two transactions will wait indefinitely for each other to release their respective locks. ◇

A deadlock is a *permanent* phenomenon. If one exists in a system, it will not go away until outside intervention takes place. There are three known methods for dealing with deadlock: *prevention, avoidance and resolution.*

3.8.2.2 Deadlock prevention

In this case the scheduler will check a transaction when it is first initiated and will not permit it to proceed if it might cause a deadlock. To perform this check, the scheduler will require that the transaction predeclare all data items to be used. The transaction manager will reserve all data items for a certain transaction before it is allowed to proceed.

This is not a very suitable method for database environments because of the difficulty of knowing which data items will be accessed by the transaction. Access to certain items may depend on the values of other items read. This method limits concurrency drastically and also requires enormous overhead in checking all transactions. On the other hand,

these systems require no run-time support, which reduces the overhead. In this system no transactions will be aborted or restarted either which makes it the ideal method for systems which have no facility for undoing processes.

3.8.2.3 Deadlock avoidance

There are two ways to accomplish deadlock avoidance; one is to employ concurrency control techniques that will never result in deadlock and the other is to require that schedulers detect potential deadlock situations in advance and prevent them from occurring.

The simplest way to avoid deadlock is to order the resources and insist that processes request access to resources in that order.

Timestamping — discussed in section 3.8.1 — also prevents deadlock from happening, as no locks are involved. Here we resolve a deadlock situation by aborting transactions with higher (or lower) priorities. Well known algorithms that use this approach are the *Wait-Die* and *Wound-Wait* algorithms [Özs91]. These algorithms are based on the assigning of timestamps to transactions. Wait-Die is a non-preemptive algorithm in that if the lock request of T_i is denied because the lock is held by T_j, it never preempts T_j. The rule is:

Definition 3.25 — *Wait-die rule*

> If T_i requests a lock on a data item that is already locked by T_j, T_i is permitted to wait if and only if T_i is older than T_j. If T_i is younger than T_j, then T_i is aborted and restarted with the same timestamp. (i.e. if $ts(T_i) < ts(T_j)$ then T_i waits else T_i dies). T_i will be restarted later with the same timestamp.
>
> This algorithm will cause an older transaction to wait longer and longer as it gets older. □

A preemptive version of the same idea is the Wound-Wait algorithm, which can be stated as follows:

Definition 3.26 — *Wound-wait rule*

> If T_i requests a lock on a data item that is already locked by T_j then T_i is permitted to wait if it is younger than T_j; otherwise T_j is aborted and the lock is granted to T_i. i.e. if $ts(T_i) < ts(T_j)$ then T_j is wounded, else T_i waits. □

By contrast to the Wait-Die algorithm, the Wound-Wait algorithm prefers the older transaction to never wait for a younger one. In both algorithms, the younger transaction is aborted. These algorithms are more suitable than prevention schemes for database applications. The only drawback is that there is considerable overhead involved. Other algorithms which prevent deadlock and do not require timestamps are the *No-waiting* and *Cautious waiting* algorithms [Els94].

Definition 3.27 — *No-waiting algorithm*

If a transaction is unable to obtain a needed lock, it immediately aborts and is restarted after a certain time delay without checking whether a deadlock will actually occur or not. Because of the constant aborting and restarting, the cautious waiting scheme was proposed to reduce the problem. □

Definition 3.28 — *Cautious-waiting algorithm*

— If T_i needs to wait for a data item held by T_j, the rule is as follows: if T_j is not blocked then T_i is blocked and allowed to wait, otherwise abort T_i.

This scheme is deadlock free. Consider the times that a transaction T_i becomes blocked as $b(T_i)$. Then, if the two transactions T_i and T_j both become blocked, and T_i is waiting on T_j then $b(T_i) < b(T_j)$, since a transaction can only wait on an unblocked transaction. Hence the blocking times form a total ordering on all blocked transactions, so no deadlock cycle can be formed. □

3.8.2.4 Deadlock detection and resolution

Another way to deal with deadlock is to detect it when it happens and then deal with it. This solution works very well in a system where most of the transactions are independent and the possibility of interference is remote. If deadlock is detected, one of the transactions involved will be rolled back and will have to execute again from the beginning. *Detection* is done by checking for deadlock situations. *Resolution* is accomplished by the selection of one or more victim transactions that will be preempted and aborted in order to break the deadlock cycle. The choice of which victim to choose can be affected by the following [Özs91]:

1. The amount of effort that has gone into a transaction. This will be lost if we abort.
2. The cost of aborting a transaction. This generally depends on the number of updates already performed.
3. The amount of effort it will take to finish the transaction. The scheduler does not want to abort a transaction that is almost finished. It will attempt to predict the behaviour of the transaction from the transaction type for example.
4. The number of cycles that contain the transaction. Since aborting a transaction breaks all cycles that it is involved in, it is best to abort the transaction that is part of more than one cycle.

3.8.2.5 Livelock

A transaction is in a state of livelock if it cannot proceed for an indefinite period of time while other transactions proceed normally. This may occur if the waiting scheme for locked items is unfair, giving priority to certain processes. The standard cure for this problem is to ensure that the priority scheme is fair. One such scheme is the *First Come First Served* queue; transactions can lock an item in the order in which transactions request the item.

Another scheme would increase the priority of a transaction the longer it waits for a certain data item lock. The transaction will eventually have the highest priority and succeed [Els94].

3.8.2.6 Starvation

Another problem which is similar to livelock is *starvation* which happens if the same transaction is repeatedly chosen to be rolled back in a deadlock situation and it never gets a chance to complete. The exact mix of transactions that would cause an intolerable level of restarts is an issue that remains to be studied [Özs91]. The wait-die and wound-wait schemes avoid starvation.

3.8.3 Optimistic methods

These methods are based on the premise that conflict is rare and that the best approach is to allow transactions to proceed unhindered and only check for conflicts when a transaction wishes to commit. Then if there is a conflict, the transaction is restarted.

The execution of any operation of a transaction follows the sequence of phases: validation(V), read(R), computation(C), write(W). Optimistic algorithms delay the validation phase until just before the write phase. Thus an operation submitted to an optimistic scheduler is never delayed. The read, compute and write operations of each transaction are processed freely without updating the actual database. Each transaction initially makes local copies of the data items and updates these local copies. The validation phase consist of checking if these updates would maintain the consistency of the database. If the answer is affirmative, the changes are written to the actual database. Otherwise the transaction is aborted and has to restart.

It is possible to have locking-based optimistic concurrency algorithms but the original algorithms were based on timestamp ordering. We will outline the latter. In this algorithm timestamps are associated only with transactions and not with data items. Timestamps are also only assigned at the time of their validation step and not at transaction initiation. This is because they are only needed at the validation stage. The algorithm has the following rules:

Definition 3.29 — *Optimistic concurrency control rules*

1. *Rule 1* — If all transactions T_k, where $ts(T_k) < ts(T_j)$ have completed their write phase before T_j has started its read phase, the validation succeeds, because transaction executions are in serial order.
2. *Rule 2* — If there is any transaction T_k such that $ts(T_k) < ts(T_j)$ which completes its write phase while T_j is in its read phase, the validation succeeds if $WS(T_k) \cap RS(T_j) = \emptyset$.
3. *Rule 3* — If there is any transaction T_k such that $ts(T_k) < ts(T_j)$ which completes its read phase before T_j completes its read phase, the validation succeeds if $WS(T_k) \cap RS(T_j) = \emptyset$ and $WS(T_k) \cap WS(T_j) = \emptyset$.

[Özs91] □

Rule 1 indicates that transactions are executed serially in their timestamp order. Rule 2 ensures that none of the data items updated by T_k are read by T_j and that T_k finishes writing its updates into the database before T_j starts writing. Rule 3 is similar to Rule 2, but does not require that T_k finish writing before T_j starts writing. It simply requires that the updates of T_k not affect the read phase or the write phase of T_j.

These algorithms allow a higher level of concurrency than timestamping and locking and this algorithm works very well when transaction conflicts are rare. A major problem is the high storage cost. To validate a transaction, the optimistic mechanism has to store the read and the write sets of several other terminated transactions [Özs91].

3.8.4 Serialization graph method

While most concurrency control methods do not test for serializability, the serialization graph method does test for conflict serializability of a schedule.

A serialization graph(SG) is a directed graph $G = (\mathcal{N}, \mathcal{E})$ that consists of a set of nodes $\mathcal{N} = \{T_1, T_2, ..., T_n\}$ and a set of edges $\mathcal{E} = \{e_1, e_2, ..., e_m\}$. There is one node in the graph for each transaction T_i in a schedule. Each edge e_i in the graph is of the form $(T_j \rightarrow T_k)$, $1 \leq j \leq n$, $1 \leq k \leq n$, where T_j is the starting node of e_i and T_k is the ending node of e_i such that one of the operations in T_j appears in the schedule before some conflicting operation in T_k.

It can be shown [Ber87] that the acyclicity of the SG is a necessary and sufficient condition to guarantee conflict serializability since a topological sort of the graph provides an ordering that corresponds to an equivalent serial execution.

Elmasri *et al* [Els94] give an algorithm for testing conflict serializability of a schedule using a serialization graph. Bernstein *et al* [Ber87] discuss serialization graphs in great detail and can be consulted for a comprehensive treatment of this topic.

Concurrency Control Method	Locks	Timestamps	Other
Conservative Schedulers	2PL Strict 2PL	Value Date	
Aggressive Schedulers	Wait Die Wound Wait	Basic TO Strict TO Multiversion TO	
Certifiers			Optimistic Concurrency Control SGT

Figure 3.14: Classification of concurrency control schemes

[Geo90]

3.8.5 Value date methods

The concurrency control method here is based on a value date [Lit89]. In executions generated by a value date scheduler, each transaction is allocated a value date which is an over-estimation of the termination time of the transaction. When a conflict between two transactions occurs, the one with the later value date will be delayed [Elm87].

3.9 Summary

The concept of a transaction has been defined, concurrent execution of transactions detailed and the problems inherent in this outlined. The concept of serializability theory has been introduced and how this relates to correctness in a database system has been elaborated upon. Recoverability of schedules has been defined and how to deal with deadlock situations has been explained. These concepts have been used to define concurrent transaction management concepts in multidatabases in chapters further on. Finally, various concurrency control methods were discussed. By way of summary we present a classification of these methods in Figure 3.14 [Geo90].

Chapter 4

Transaction Management

This chapter discusses the problems inherent in multidatabase transaction management, global concurrency control and global deadlock. The formal transaction model which was introduced in Chapter 3 is extended to include multidatabase concepts. The functions of the global transaction manager are outlined and past research into transaction management, global concurrency control and global deadlock management schemes which appear in the literature is evaluated.

4.1 Transaction Management in Multidatabase Systems

The major task of transaction management in a multidatabase environment is as follows [Bri92]:

> "Ensuring the global consistency and freedom from deadlocks of the multidatabase system in the presence of local transactions (i.e. transactions executed outside of the multidatabase system control), and in the face of the inability of local DBMSs to coordinate execution of multidatabase transactions (called global transactions), under the assumption that no design changes are allowed in local DBMSs."

The majority of research in multidatabases has concentrated on data models and schema integration and not much research has been done into the problem of transaction management in multidatabase systems.

An MDB architecture involves a number of database systems, each with its own *local transaction manager* (LTMs) and a multidatabase software layer (MDMS) on top. The local database systems may not be aware of the existence of other participating local database systems, and thus may not be able to communicate with them. As a result, local DBMSs may be incapable of ensuring that the concurrent execution of local and global transactions preserve database consistency. Thus a software module, referred to as the *global transaction manager* (GTM), is built on top of the existing database systems in order to coordinate the

execution of global transactions in a MDB environment [Bre95]. The GTM must provide the following functionality in a multidatabase [Kan93]:

1. Scheduling — The GTM must control the submission of global transaction operations to the appropriate LTMs such that the global history is serializable. The main objectives of a multidatabase scheduler are to ensure [Rus92]:

   - *Correctness*: The scheduler must take the execution dependencies that have been defined for the multidatabase transaction into account, as well as the constraints imposed by the global concurrency control method which specifies allowable interleavings of subtransactions.
   - *Safety*: The scheduler must guarantee that the multidatabase transaction will terminate in one of the specified acceptable termination states. Before executing a transaction, the scheduler should examine it to see whether it satisfies this requirement. If it is unable to determine the safety of a transaction, it should reject the transaction without attempting to execute it.
   - *Optimal scheduling policy*: A multidatabase scheduler should achieve an acceptable termination state in the "optimal" way. This could vary from application to application. One possibility is to define it as achieving the goal in the shortest possible time. Alternatively, we may associate a cost function with the execution of each subtransaction. The objective of the scheduler would then be to execute the transaction with the minimum cost.
   - *Handling of failures*: A scheduler should be able to reach an acceptable termination state even in the case of a failure. The scheduler must use stable storage to log all the information about its state so that it could recover if need be.

2. Atomic commitment protocol — The GTM is assumed to perform an atomic commitment protocol for supporting the atomicity of global transactions. The LTM is not assumed to participate directly in this protocol.

3. Recovery management — The GTM must restore the multidatabase system to a consistent state following a failure of any type. Due to the autonomy of the local database systems, the GTM does not have access to the local log. Thus the GTM must maintain a separate log for its own recovery purposes.

Scheduling will be discussed in this Chapter. Chapter 6 will handle the recovery management function while the atomic commitment protocol will be discussed in chapters 5 and 6.

The computation model which will be used in the rest of this dissertation is that of Barker [Bar90] and is shown in Figure 4.1. The general computational model can be summarized as follows:

A global transaction is managed by the MDMS layer which parses it into a set of *global subtransactions.* Global subtransactions consist of those operations of the global transactions which belong to a particular local database. Each subtransaction is submitted to a local database system. Local DBMSs are responsible for the concurrent execution of both the global subtransactions and the local transactions submitted to them. The synchronisation of global transactions is the responsibility of the MDMS layer [Bar90].

4.1.1 The role of the global transaction manager

A series of conditions can be identified [Özs91, Gli86, Bar90] that specify when global transactions can safely update in a multidatabase system. These conditions are helpful in determining the minimal functionality required of the various transaction managers. The conditions are:

1. The individual database managers must guarantee *local synchronization atomicity.* This means that local transaction managers are simply responsible for the correct execution of the transactions on their respective databases. Each local recovery manager is responsible for maintaining that its schedule is serializable and recoverable. These schedules are made up of global subtransactions as well as local transactions. The local DBMS therefore accepts a transaction and executes it till termination.

2. Each LTM maintains the *relative execution order* of the subtransactions as determined by the GTM. The GTM is responsible for coordinating the submission of the global subtransactions to the LTMs and coordinating their execution. This is complicated by the fact that in a MDMS, the GTM cannot communicate directly with schedulers at local sites. This is because:

   - firstly, the individual local nodes do not necessarily know how to communicate in a distributed environment; and
   - secondly, GTMs already have difficulty scheduling transactions across multiple sites and it may not be feasible for them to get even more involved with transaction scheduling across multiple DBMSs at one site. This would entail a GTM sending a global subtransaction to another global transaction manager at another site and expecting it to coordinate the execution of the global subtransaction. The GTM at the other site may then further decompose the transaction into global subtransactions, depending on the organization of the local databases at its site.

   The GTM is therefore only responsible for the serializability of the global transaction execution histories (a history is the global version of the schedule concept as defined in the previous chapter).

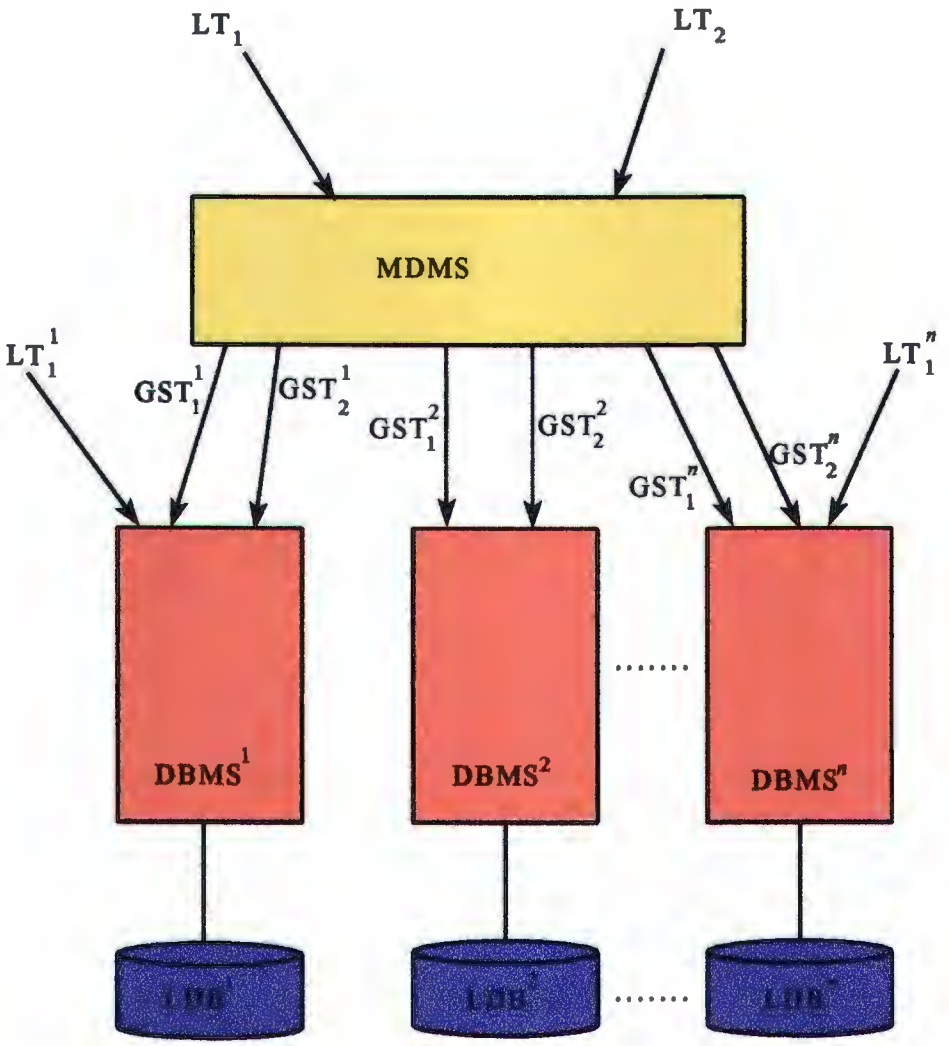


Figure 4.1: Depiction of the computational model

[Bar90, p29]

3. The GTM is also responsible for dealing with *global deadlocks* that occur amongst global transactions and should provide the means to recover from any type of system failure. This ordering is fairly simple to maintain if the GTM waits for the result of one subtransaction before submitting the next one but of course the degree of concurrency in this case is very low.

4. The GTM should guarantee the *ACID properties* of global transactions, even in the presence of local transactions that the GTM is not aware of. Atomicity in a multidatabase has two properties [Gli86]:

   - *Failure atomicity* means that a global transaction behaves as though it executes to completion or not at all. In other words, if one of the local systems crashes in the middle of executing a transaction, intermediate results will not be left in the multidatabase.
   - *Synchronization atomicity* guarantees that during the normal concurrent execution of global transactions the results will be the same as if the transactions were executed in some serial order.

   In a homogenous distributed database system, atomicity of transactions is guaranteed by an *atomic commit protocol* such as the two-phase-commit (2PC) protocol[1]. This protocol requires participating local sites to provide a *prepare to commit* command in their interface. When the local DBMS receives and acknowledges a prepare-to-commit command, it makes a promise to the GTM that it will commit its work if so requested by the GTM [Bre95].

   However, this may not be possible in a multidatabase system. If participating local database systems in a multidatabase system *do* all export a prepare-to-commit command (so that it is possible to use the 2PC protocol), then they lose some of their execution autonomy since the individual DBMSs are no longer free to make decisions regarding resources held by the global transaction at the local database system. As stated before, the atomicity issue in multidatabases is not a trivial one.

5. Global transactions cannot be split up and submitted concurrently to the same local database system.

6. The MDMS must be able to identify all objects referenced by all global transactions.

4.1.2 Extending the formal transaction model to include multidatabase concepts

In Chapter 3 we introduced a basic transaction model which we now extend to include multidatabase concepts.

[1] See Appendix C

Local and global transactions will be formally defined with respect to the data items they access as well as their mode of operation.

Definition 4.1 — *Local database*

Each of the autonomous databases that make up a multidatabase is called a *local database* (LDB). The set of data items stored at a local database system LDB^i, the local database at site i, is denoted by $\mathcal{LDB}^i$.

The set of all data in the multidatabase can be defined as:

$$\mathcal{MDB} = \bigcup_i \mathcal{LDB}^i$$

[Bar90] □

Definition 4.2 — *Local transaction*

A transaction T_i submitted to DBMS j, (denoted by $DBMS^j$) is a *local transaction* (denoted by LT_i^j) on $DBMS^j$ if $\mathcal{BS}_i \subseteq \mathcal{LDB}^j$ where $\mathcal{LDB}^j$ is the local database managed by $DBMS^j$.

We denote the set of all local transactions at LDB^j by $\mathcal{LT}^j = \bigcup_i LT_i^j$. The set of all local transactions in a multidatabase system is $\mathcal{LT} = \bigcup_k \mathcal{LT}^k$.

[Bar90] □

Definition 4.3 — *Global transaction*

A transaction is a *global transaction* (GT_i) iff:

1. $\nexists \mathcal{LDB}^j$ such that $\mathcal{BS}_i \subseteq \mathcal{LDB}^j$ or
2. GT_i is submitted to $DBMS^k$ but $\mathcal{BS}_i \subseteq \mathcal{LDB}^r (k \neq r)$.

We will let $\mathcal{GT}$ denote the set of all global transactions in the multidatabase, i.e.

$\mathcal{GT} = \bigcup_i GT_i$.

[Bar90] □

Item (1) states that global transactions submitted to the MDMS access data items stored in more than one database. In other words, if the transaction accesses data items contained within a single database, the transaction is not a global transaction but is referred to as a

local transaction. Item (2) represents the case where a user working on one database system requires access to the data stored and managed by another DBMS in another database system. A global transaction GT is parsed into a set of *global subtransactions* GST_1, GST_2,... GST_n which are subsequently submitted to local database systems for execution. Thus a global transaction is executed as a set of subtransactions that execute on a number of local DBMSs. Global subtransactions are defined in terms of the data items referenced, and with respect to the global transaction creating them.

The decomposition of GT is mainly based on the sites at which the data items reside. The subtransactions generated should satisfy the following conditions [Geo90]:

1. There is at most one global subtransaction per LDB for each global transaction.
2. In the case where the GTM submits operation by operation of a subtransaction to the local database, transactions executed at local databases that do not support a prepare to commit state must perform a handshake after each database operation so that database operations are totally ordered. In the case where the GTM submits service requests, this would not be a requirement of the subtransaction.
3. Subtransactions include "send" and "receive" operations whenever data movement and task synchronization between the MDMS and the subtransactions are necessary. It is assumed that subtransactions perform a handshake after each receive operation. The send and receive can be modeled by write and read operations issued to data items no other transaction accesses.

Definition 4.4 — *Global subtransaction*

A global subtransaction submitted to $DBMS^j$ on behalf of a global transaction GT_i (denoted GST_i^j) is a transaction where:

1. $\Sigma_i^j \subseteq \Sigma_i$ and
2. $\mathcal{BS}_i^j \subseteq \mathcal{LDB}^j$ where $\mathcal{BS}_i^j$ is the base-set for GST_i^j .

The set of all global subtransactions submitted to a particular local database $DBMS^k$ is denoted by $\mathcal{GST}^k$ while the set of all global subtransactions produced by a global subtransaction GT_i is denoted as $\mathcal{GST}_i$.

Therefore the set of all global subtransactions in a multidatabase system is:

$$\mathcal{GST} = \bigcup_i \mathcal{GST}_i = \bigcup_k \mathcal{GST}^k.$$

[Bar90] □

This definition formalizes the assertion that each global subtransaction executes at only one DBMS. Therefore a global subtransaction can be seen as a local transaction by the DBMS to which it is submitted.

We will now define *histories* which will be used later in defining recoverability concepts. In the previous chapter we used the concept of schedules to denote the interleaving of operations of various transactions in a single database. When we look at the interleaving of operations in a multidatabase, we will refer to schedules as histories to allow us to distinguish between the two concepts.

Definition 4.5 — *Local history*

Given a LDB^k with a set of local transactions $\mathcal{LT}^k$ and a set of global subtransactions $\mathcal{GST}^k$, a *local history* (LH^k) is a partial order $LH^k = (\Sigma^k, \prec^k_{LH})$ where:

1. $\Sigma^k = \bigcup_j \Sigma^k_j$ where Σ^k_j is the domain of transaction $T_j \in \mathcal{LT}^k \cup \mathcal{GST}^k$ at LDB^k,
2. $\prec^k_{LH} \supseteq \bigcup_j \prec^k_j$ where $\prec^k_j$ is the ordering relation for the transaction T_j at LDB^k, and
3. for any two conflicting operations $p, q \in LH^k$, either $p \in \prec^k_{LH} q$ or $q \in \prec^k_{LH} p$.

[Bar90] □

The collection of global subtransactions at each local database is sufficient to describe the ordering of global transactions. We will define the global subtransaction history next as a subset of the local site histories by restricting Definition 4.5.

Definition 4.6 — *Global subtransaction history*

The global subtransaction history of LDB^k is described by the partial order $GSH^k = (\Sigma^k_{GSH}, \prec^k_{GSH})$ where:

1. $\Sigma^k_{GSH} = \bigcup_j \Sigma^k_j$, where Σ^k_j is the domain of transaction $T_j \in \mathcal{GST}^k$ and
2. $\prec^k_{GSH} \subseteq \prec^k_{LH}$.

[Bar90] □

A global transaction history can be defined by global subtransaction orders at each local database. A global history is the union of global subtransaction histories at each participating local database.

Definition 4.7 — *Global history*

A global history $GH = (\Sigma_{GH}, \prec_{GH})$ is the union of all global subtransaction histories:

1. $\Sigma_{GH} = \cup_k \Sigma^k_{GSH}$,

2. $\prec_{GH} \supseteq \cup_k \prec^k_{GSH}$, and
3. for any two conflicting operations $p, q \in GH$, either $p \prec_{GH} q$ or $q \prec_{GH} p$.

[Bar90] □

The final definition describes the history of a multidatabase execution. Essentially the MDB history is fully described by the combination of local site histories (Definition 4.5 and Definition 4.7).

Definition 4.8 — *MDB history*

A history of a multidatabase (denoted by MH) consisting of n local histories and a global history (GH) can be described as a tuple $MH = \langle \mathcal{LH}, GH \rangle$ where $\mathcal{LH} = \{LH^1, LH^2, ..., LH^n\}$.

[Bar90] □

4.2 Transaction Management Approaches

This section introduces the research done into transaction management in our selected core group of schemes.

4.2.1 Barker & Özsu's basic MDB model

Barker *et al* [Bar90, Özs91] propose a very basic MDB model which is illustrated in Figure 4.2. This model serves to give a good basic understanding of multidatabase system architecture.

The MDB consists of various local database systems each having its own DBMS, each of which manages a different local database system. The MDMS provides a layer of software that runs on top of these local database systems and allows users to access various database systems. Each DBMS has its own transaction processing components: the local transaction manager, the local data manager, and a local scheduler. The MDMS is simply seen as another user from which transactions are received and to whom results are presented. The MDMS layer consists of a global transaction manager, a global scheduler and a global recovery manager. Barker uses the notion of *m-serializability* (see Synopsis 4.22, section 4.4.2.1) for correctness and uses multidatabase serializability graphs to maintain global concurrency control [Bar90].

4.2.2 Pu's hierarchy of superdatabases

Pu [Pu88] describes multidatabases in terms of a *hierarchy of superdatabases* as illustrated in Figure 4.3. Each participating database system (called an element database) can be pictured

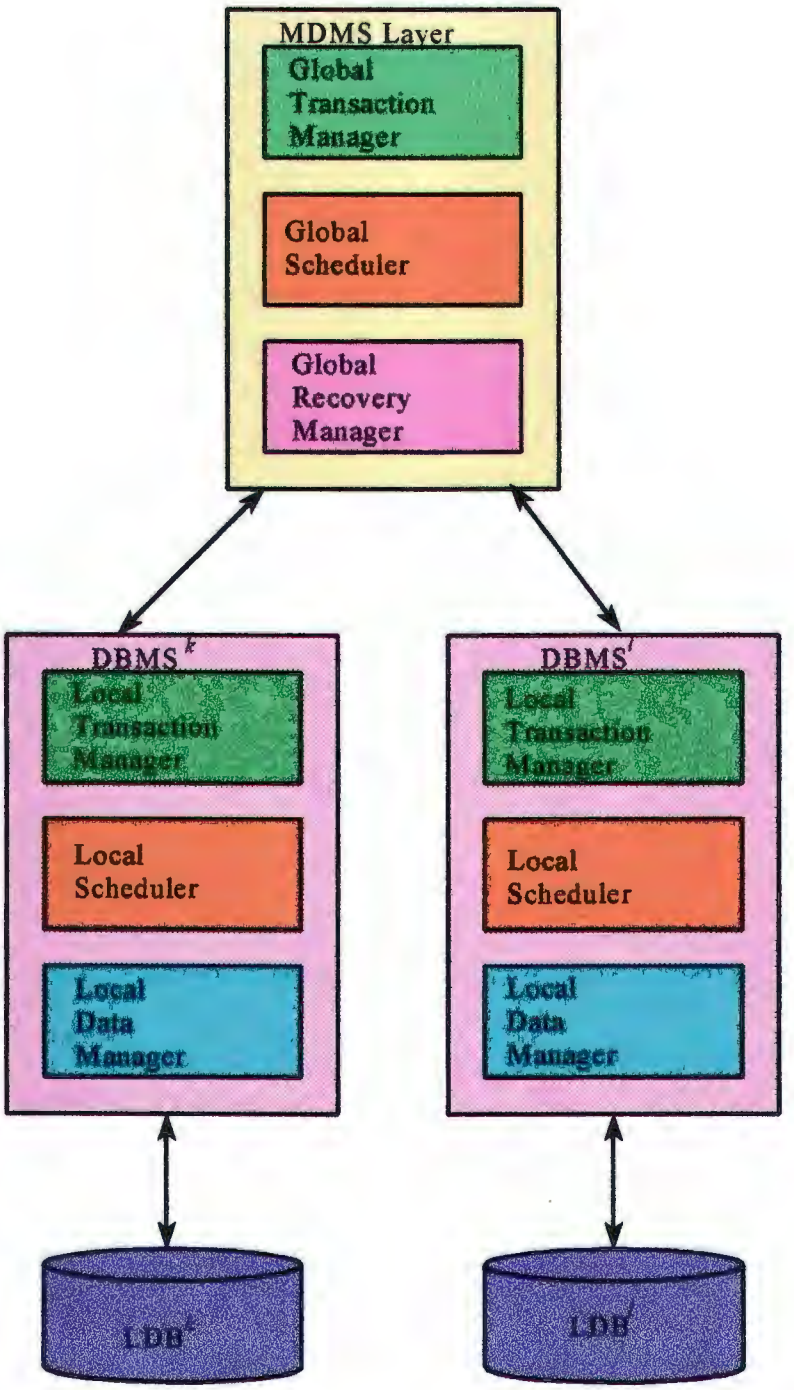


Figure 4.2: Components of an MDB in Barker & Özsu's model

[Bar90, p10]

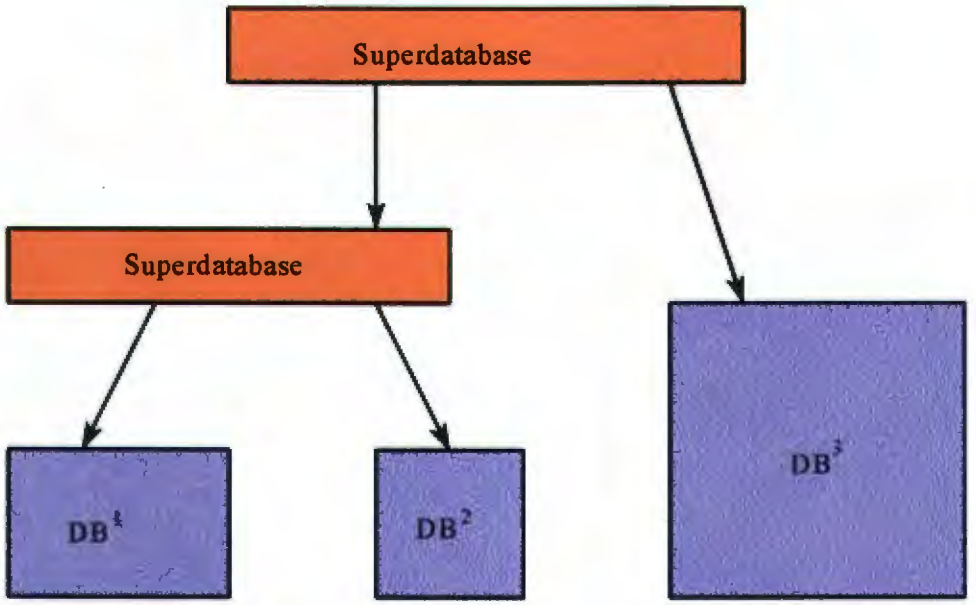


Figure 4.3: Pu's multidatabase transaction processing model

[Pu88, p146]

as the *leaves* on a tree and each internal node as a superdatabase that manages the element databases in a hierarchical structure. Each element database operates independently but global activities are managed at the node level of the tree. Transactions that cross multiple element databases are called supertransactions and are posed against a superdatabase.

Superdatabases utilize serializability as a transaction correctness criteria. Serializability is ensured by having each element database provide the superdatabase with information about the ordering of its local transactions. Pu claims this is not necessary when element databases provide strict schedules. Serial orderings of each local transaction at the local site is provided by order-elements (O-elements). The O-element for each transaction is passed to the superdatabase, after which the ordering of supertransactions can be determined. When the ordering has been worked out, it is analyzed and if it is serializable, the supertransaction can commit, otherwise each subtransaction is aborted [Bar90]. This architecture was implemented at Columbia University — called the Harmony system [Pu91b].

There are some problems with Pu's approach. Formation of the O-element ordering requires that the local site keep the superdatabase informed of decisions made locally. This violates local autonomy. The superdatabase, and not the local databases, makes arbitrary decisions about whether transactions may commit, which also violates autonomy. Pu has also not addressed the problem of crash recovery. [Bar90]

4.2.3 Breitbart *et al*'s work

Breitbart *et al* [Bre88, Bre86, Bre85, Bre87] have done research into many aspects of multidatabase systems. We will only discuss their research into transaction management here. They assume that data can be replicated.

Breitbart *et al* initially used the same multidatabase architectural model as Barker *et al* (in Figure 4.2) but had a different approach to defining global transactions. Breitbart *et al* defined a global database as a triple (D, S, f) where D is a set of global data items, S is a set of sites and f is a function: $f : D \times S \rightarrow (0,1)$ such that $f(x,i) = 1$ means that data item x is available at site i. If $f(x,i) = f(x,j) = 1$ where $i \neq j$, then x is replicated at i and j. Breitbart *et al* use serializability as a correctness criterion and serialization graphs to determine when a history is serializable.

In this model global operations are mapped onto local operations and considered as a single local schedule to determine whether schedules are serializable. The most recent work of these researchers [Bre90a] addresses the problems of reliability. Their approach is based on splitting up data into mutually exclusive groups of *locally* and *globally updateable* data.

Breitbart *et al* proposed solutions to both concurrency and reliability problems in multidatabases. Breitbart *et al*'s reliability proposal does not provide for full autonomy of the local databases and splitting up data into locally and globally updateable groups violates local autonomy [Bar90].

In later work on the transaction management problem Breitbart *et al* extended the basic transaction processing model to a client-server type model which uses *servers* at local sites [Bre95, Geo90, Meh93, Meh92c, Meh92b, Bre90a]. They propose a transaction processing model where the software module includes a *global transaction manager* (GTM) at the multidatabase site, and a set of *servers*, one associated with each participating database system. The multidatabase transaction processing model is illustrated in Figure 4.4

In this basic transaction processing model, each global transaction submits its *read/write* operations to the GTM. For each submitted operation, the GTM then determines whether to submit the operation to local sites, or to delay it, or to abort the transaction. If the operation is to be submitted, the GTM will select a local site (or a set of sites) where the operation should be executed [Bre95].

The GTM submits operations to the local DBMSs through the server, which acts as a liaison between the GTM and the local DBMS. Operations belonging to a single global subtransaction are submitted to the local DBMS by the server as a single local transaction.

There are two possible ways for these servers to be utilized. The exact way in which the GTM and the local DBMS interact depends on the schema exported by the local DBMS.

- One possibility is for the DBMS to accept individual read and write operations. In this case, before the server initiates actions on behalf of a global subtransaction, it starts a new local transaction by issuing a *begin transaction* operation to the local DBMS. The DBMS returns a transaction identification that is used in subsequent actions

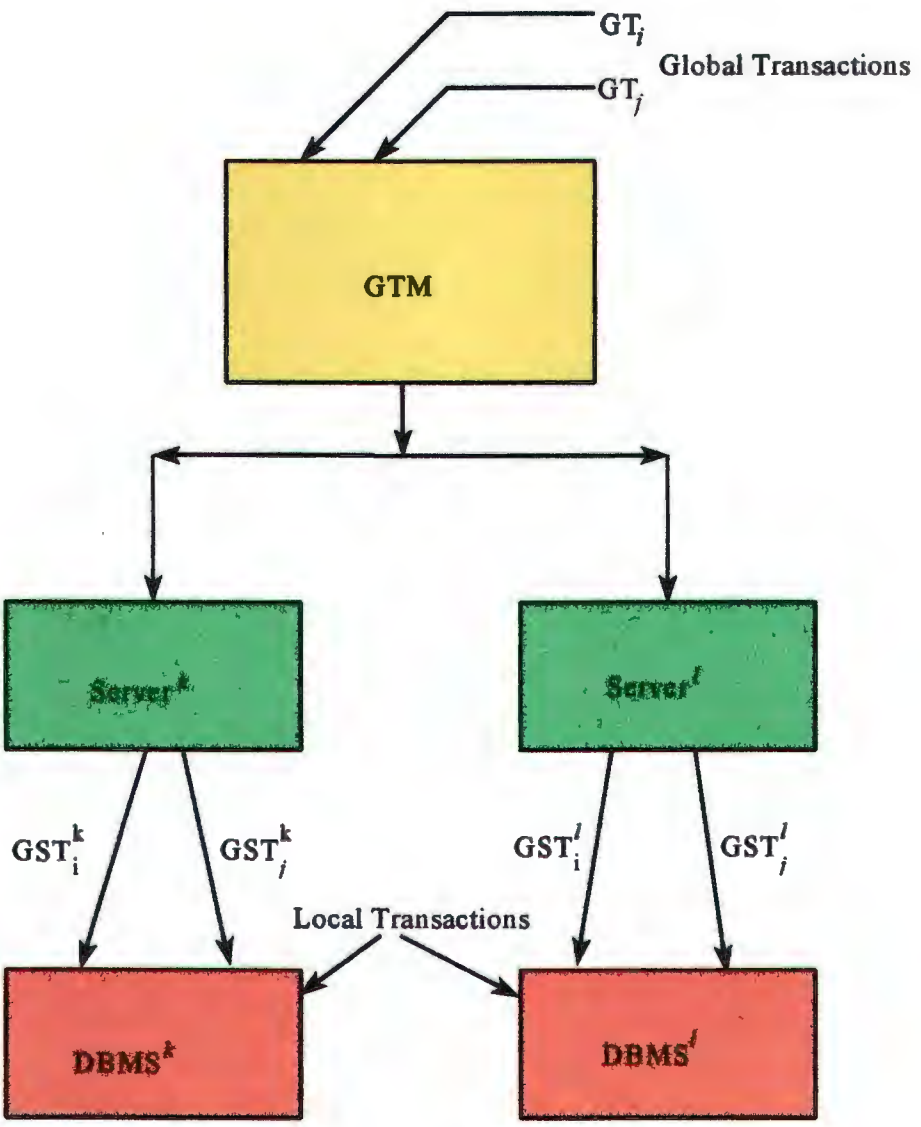


Figure 4.4: Breitbart *et al*'s multidatabase transaction processing model

[Bre95, p576]

of the subtransaction. After each read and write action is submitted, the DBMS acknowledges execution (and if a data item was read, includes the value obtained). When the GTM wishes to commit, it issues (via the server) a commit command. This type of scheme violates local autonomy and for this reason is not ideal.

- Another possibility is for the GTM to send service requests to the individual DBMSs such as 'reserve a seat on flight x'. In this case the server can only call on a set of predefined services; and each call represents an implicit local transaction [Bre95]. This scheme does not violate local autonomy and therefore has more to recommend it.

Variations of these two server schemes are also possible and have been discussed in the literature [Bre92c]. It all depends on whether the individual DBMS is prepared to participate in a global commit protocol or not.

4.2.4 Elmagarmid *et al*'s work

Elmagarmid *et al* have done much work in the area of transaction management in multidatabase systems [Elm88, Elm87, Elm86]. Their initial work focused on characterizing the problem and proposed a number of pragmatic approaches. The research proposed maintaining a high level of local site autonomy while using serializability as a correctness criterion. This autonomy is maintained by adding software to local sites to ensure serializable schedules at each local site. The monitoring and submitting software is known as a stub process [Bar90].

Elmagarmid & Helal proposed *weak-stub/strong-stub* processes for ensuring correct execution of global transactions [Elm88]. The approach is to submit a subtransaction as a weak-stub process if it could be aborted by the local DBMS and as a strong stub once the subtransaction had to commit. Unfortunately, a situation could arise where the local DBMS could consistently refuse to commit the subtransaction [Bar90].

Du & Elmagarmid [Du89] recognized the difficulties of using serializability as a correctness criterion where a problem arises due to indirect conflicts occurring between subtransactions and local transactions. This led to a notion of *quasi-serializability* (see Synopsis 4.18, section 4.4.2.1) [Du89]. Du & Elmagarmid [Du89, Elm90a] later merged the pragmatic approach with the quasi-serializability correctness criterion [Bar90].

4.2.5 Chen *et al*'s distributed MDMS

Chen *et al* [Che93] extended Elmagarmid *et al*'s work by proposing a *distributed* MDMS which is not vulnerable to failures. The regular architecture we have described up to now has a central node which, if it fails, incapacitates the whole system.

The MDMS described by Chen *et al* consists of a *global transaction manager* and a set of *interfaces* located at each site. The GTM controls execution of all MDMS transactions. For

each MDMS transaction T_i a GTM process GTM_i, which is responsible for the consistent and reliable execution of T_i, is issued. GTM_i is therefore coincident with the life cycle of T_i. The interface accepts and schedules the execution order of subtransactions on the local system where it resides and creates a *server procedure* for each subtransaction in the system. The server is coincident with the lifecycle of the subtransaction.

The architecture of Chen *et al*'s model is illustrated in Figure 4.5. Before a transaction is executed, it requests all corresponding MDMS interfaces to arrange the scheduling order of its subtransaction on the corresponding LDBSs so as to prevent any MDMS inconsistencies its execution may cause. When executing a global transaction, a GTM process only interacts with relevant MDMS interfaces, without the need to communicate with other GTM processes.

Each GTM process can thus run independently and the GTM is made a distributed entity. No assumptions are made about the LDBSs and the autonomy of the LDB is maintained.

4.2.6 Kang & Keefe's decentralized GTMs

Kang & Keefe [Kan93] propose a multidatabase model where the GTM is totally decentralized. Their model caters for multiple versions. The basic model is illustrated in Figure 4.6. This scheme differs from Chen *et al*'s scheme [Che93] because in Chen *et al*'s scheme the GTM is located at the site where the transaction is submitted and *that* GTM communicates with interfaces at all the other local database sites, whereas Kang & Keefe's scheme locates a GTM at each local site which does all the work — there are thus no servers.

At each site there is a GTM accepting global transactions from users and receiving subtransactions from other GTMs via the network. The GTM maintains a global directory and therefore can determine the appropriate sites at which a global transaction will execute.

All operations executed at a site from the same global transaction constitute a subtransaction. The LTM does not distinguish between local and global transactions. Only the LTM can access the local database. The LTM is responsible for the concurrency control and recovery of the database while the GTM must maintain the consistency of the multidatabase.

4.2.7 Garcia-Molina & Salem's sagas

Garcia-Molina & Salem [Gar87] have proposed a nested transaction model intended to deal with long lived transactions. Their model uses nested transactions called *sagas*, with only two levels of nesting. A saga is not executed as an atomic unit. This means that the results of a subtransaction's execution are visible as soon as it commits and not only after commitment of the entire saga. Sagas are written so that they are interleavable with any other transactions which makes concurrency control at the saga level unnecessary. Because of this design factor, the introduction of local transactions does not cause any

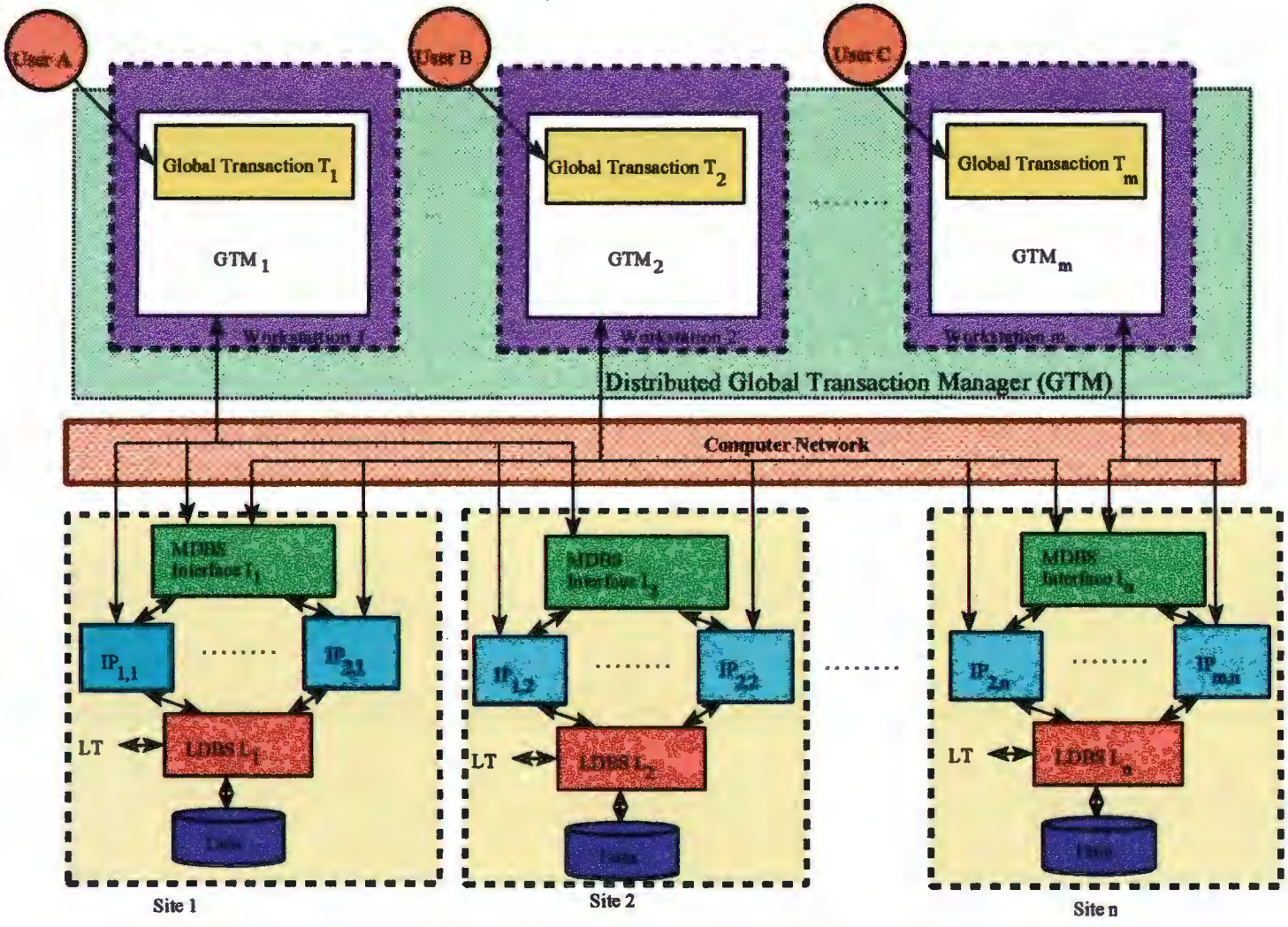


Figure 4.5: Chen, Bukhres & Sharif-Askary's MDMS architecture

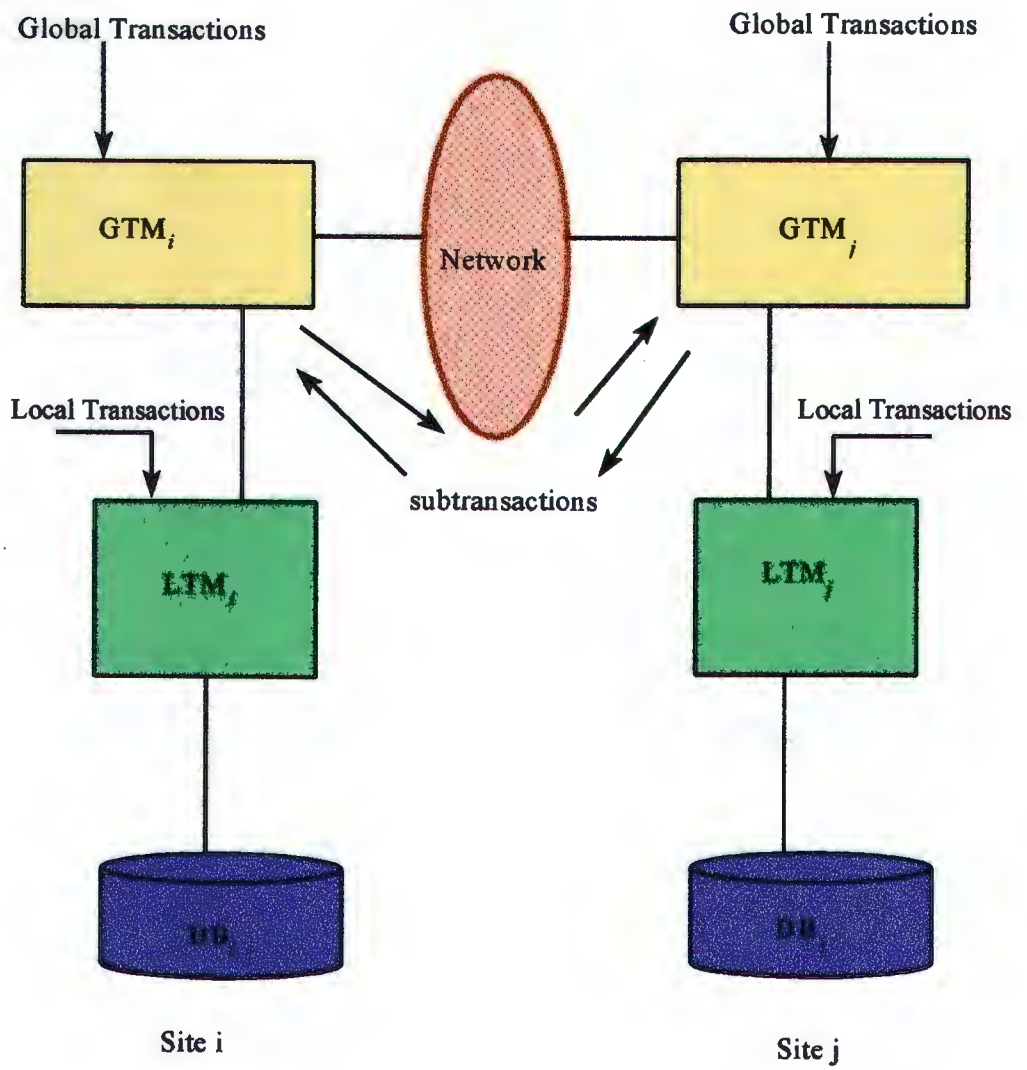


Figure 4.6: Kang & Keefe's multidatabase architecture

[Kan93, p458]

concurrency control problems. In this model two assumptions are made which make this approach unsuitable in multidatabase systems. Firstly, the model is not applicable to all multidatabase environments since it may be too restrictive to require that sagas be interleavable with other transactions. Secondly, it may not always be possible to write the compensating transactions that this model requires. [Bar90]

4.2.8 Yoo & Kim's client server approach

Yoo & Kim [Yoo95] propose the multidatabase architecture as illustrated in Figure 4.7.

In the figure, LT_j denotes a local transaction issued and executed locally in site S_j. GT_i denotes a global transaction and GST_i^j denotes a subtransaction of GT_i submitted to $LDBMS^j$. The three components can be described [Yoo95]:

- *Multidatabase transaction manager* (MDBS_TM). The MDBS_TM submits subtransactions of a global transaction to the appropriate LDBMSs via *agents*. The MDBS_TM controls the submission of concurrent global transactions in order to achieve a correct schedule.

- *Agent*: The agent is a component of the MDMS that runs on each site, which is merely an application in a local DBMS's viewpoint. It receives operations of subtransactions from the MDBS_TM, submits them to the LDBMS and send the results to the MDBS_TM.

- *Stub*: Each stub extracts concurrency control information from the requested operations on the local database and controls operation submission to the LDBMS. The stub thus controls *all* database update operations, both local and global. The stub uses stub-level locks to control submission of operations of local transactions as well as global subtransactions.

This transaction management scheme provides a reliable transaction management mechanism which maintains global consistency in the face of failures. Yoo & Kim assume that each local database system uses 2PL as its concurrency control scheme. Yoo & Kim contend that this requirement is not restrictive because most commercial DBMSs use the 2PL protocol. [Buk93, Bre91a, Vei92, Kim93]

4.2.9 Other research

In this section we introduce other research into transaction management in multidatabase systems which does not form part of the core group but which also merits discussion.

4.2.9.1 Nodine & Zdonik's step scheme

Nodine & Zdonik [Nod94] propose a *step approach* to integrating the information in local databases into a multidatabase. This approach has been implemented in the Mongrel

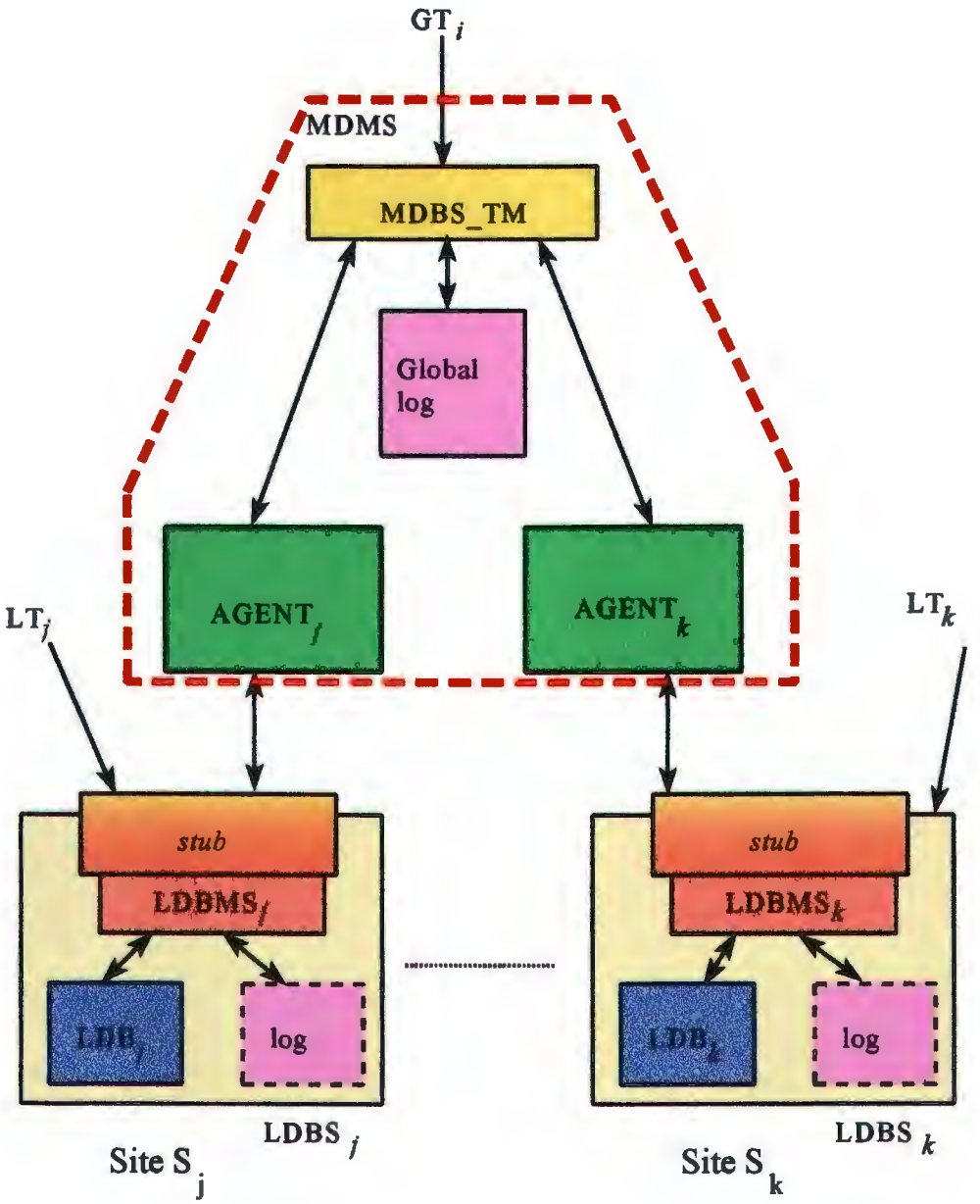


Figure 4.7: Yoo & Kim's multidatabase system architecture

[Yoo95, p56]

prototype multidatabase system.

In this model, each local database system defines an interface of function calls or steps that it is willing to provide for use by the multidatabase. These steps are collected into the local database's own *step library.* In the step library, each step is explicitly paired with its compensating[2] step. The compensating step will reverse the effects of the step on the database. The step itself is not a database operation — several operations can be grouped into a single atomic global subtransaction. The step will log information about the values of data items as it executes so that the compensating step knows the values and can reverse the actions of the step.

This step approach provides adequate information to determine whether a step is easily compensatable or likely to cause trouble in specific situations. Problems occur when interference from other transactions prevents compensation from succeeding [Nod94].

4.2.9.2 Rusinkiewicz *et al*'s flexible transactions

The concept of *flexible transactions* was proposed by Rusinkiewicz *et al* [Rus90] to extend the basic transaction model. This model allows global transactions to release atomicity by specifying subtransaction dependencies. Dependencies that can be specified between a pair of subtransactions GST_k^j and GST_k^i include:

- Positive execution dependency defines that GST_k^i cannot be executed until GST_k^j completes successfully.
- Negative execution dependency exists if GST_k^i cannot be executed till GST_k^j has been invoked and did not achieve its objectives or failed.
- Alternative dependency specifies that GST_k^i and GST_k^j accomplish the same objective.
- Compensating execution dependency specifies that GST_k^i rolls back the effects of GST_k^j if it is executed after the successful completion of GST_k^j.

Therefore, flexible transactions can be successful (achieve their objectives) even if some of their actions fail and others are never executed. The flexible transaction model also captures the temporal aspects of transaction execution because a commitment date and expiration date and a temporal predicate can be associated with each transaction. [Geo90]

4.2.9.3 Litwin & Tirri's timestamps

Litwin & Tirri [Lit89] propose the use of timestamps to determine whether transactions execute correctly. A data item is assigned a value date which indicates the time that the item was given a correct value. When transactions are created, an actual date is

[2] A compensating transaction is a transaction that is run to negate the effects of a transaction that has already run. More details are given in Chapter 6

assigned to them and if the transactions actual data and data items value dates indicate that safe access is possible, the transaction is permitted to execute. This approach delegates the synchronization of global transactions to the local transaction managers. However, it assumes that the individual DBMSs are able to compare transaction timestamps with data timestamps. This scheme violates local autonomy. [Bar90]

4.2.9.4 Georgakopoulos *et al*'s forced local conflicts

Georgakopoulos & Rusinkiewicz *et al* have worked on reliable multidatabase transactions [Geo90, Geo91b]. Their method also uses serializability and introduces a class of schedules called *rigorous schedules.* Their work is very significant because it seems to solve the problem of indirect conflicts (see section 4.3.2). Rigorousness may be too restrictive but that remains to be seen. This work also addresses atomicity of multidatabase transactions. [Bar90]

4.2.9.5 The StarGate MDMS

StarGate is an MDMS prototype which has been developed at the University of Stellenbosch [Key93]. The client-server based MDMS is also distributed in this system. It consists of four components built on top of a local database system at each local site:

- A presentation manager which accepts transactions.
- The Star server which receives transactions from the presentation manager and which creates subtransactions. This is the global component which communicates with stars and gates at other sites. The star is multithreaded and acts like a normal centralized DBMS server. Client processes connect to the star server and request a session to the global database.
- The Gate server which consists of a transaction manager, scheduler and local session manager and provides scheduling and resource control. The gate also communicates with gates and stars at other sites. The gate is multithreaded and can serve multiple clients simultaneously.
- The DBMS server communicates with the local DBMS This is a concurrent server which means that each client process spawns a dedicated server process. This server provides the user with a transparent view of any component database.

StarGate uses a commit protocol called "take-a-ticket" which ensures global serializability while also solving the indirect conflict problem. Each site will control the commitment of global transactions submitted at that site.

4.3 Integrating Various Concurrency Control Methods

Virtually all DBMSs adopt a static approach to concurrency control. When the DBMS is being designed, a number of factors are taken into consideration and on the basis of this, a single concurrency control method is adopted and built into the system [Bel92].

In a MDMS it is highly unlikely that the concurrency mechanisms supported by the local DBMSs are identical.

In order to integrate heterogeneous local concurrency control algorithms, we have to consider the following two problems [Yun93]:

1. How we can process a global transaction when it violates global serializability or has the possibility of violation of global serializability. This is referred to as the *processing problem.*

2. How we can manage an indirect conflict introduced by a local transaction. This is referred to as the *indirect conflict problem.*

The traditional approaches to integrating heterogeneous concurrency control algorithms can be classified into two groups: *bottom-up* and *top-down* approaches.

- The *bottom-up* approach collects local information from each local site at the global level and thereafter checks global serializability. This approach is *optimistic* and requires considerable time and cost for the modification of each local DBMS. This approach violates local autonomy and is therefore not ideal. Another problem is that many global transactions may have to be aborted due to violation of global serializability. Consequently, cascading rollback could be caused by aborting of global transactions.

- The *top-down* approach maintains a global serialization order at local sites, which has been determined already at the global level. This method is *pessimistic* and does not require modification of local DBMSs at all. In this respect, this approach maintains local autonomy. To determine the serialization order of the global transactions at each LDB, the MDMS must deal with both direct conflicts as well as indirect conflicts.

  In this approach, the indirect conflicts among global transactions via local transactions can not be considered at the global level because the global level is unaware of them.

  Some of the indirect conflicts may cause a discrepancy between the *execution order* of global transactions and their *serialization order.* Several solutions to the indirect conflict problem have been proposed but many of them are not satisfactory [Yun93].

Example 4.1 : Execution order and serialization order

Consider two sites with LDB^1 and LDB^2 with data items $\{a\}$ and $\{b, c\}$ respectively. Say we have the following global transactions:

$GT_1 : w_1(a); r_1(b)$

$GT_2 : r_2(a); w_2(c)$

and a local transaction:
$L_1 : w_L(b); r_L(c)$

We now have the following global transaction history:
$GH : w_1(a); r_1(b); r_2(a); w_2(c)$

and two local histories are::
$LH^1 : w_1(a); r_2(a)$
$LH^2 : w_L(b); r_1(b); w_2(c); r_L(c)$

Let $\overset{d}{\rightsquigarrow}$ and $\overset{i}{\rightsquigarrow}$ denote direct and indirect conflicts respectively.
There is a direct conflict $GT_1 \overset{d}{\rightsquigarrow} GT_2$ in LH^1 and an indirect conflict $GT_2 \overset{i}{\rightsquigarrow} GT_1$ in LH^2 because $L \overset{d}{\rightsquigarrow} GT_1$ and $GT_2 \overset{d}{\rightsquigarrow} L$ in LH^2. The execution order of LH_1 and LH_2 are both GT_1, GT_2 but an indirect conflict between GT_1 and GT_2 via L is produced in LH_2. Thus at LDB^1 the execution order (i.e. GT_1, GT_2) becomes different from the serialization order (i.e. GT_2, GT_1) in LH_2. The global history would therefore be unserializable because the serialization order is different at LDB^1 and LDB^2.

◇

The concurrency control mechanism in a multidatabase has to be able to synchronize global transactions with purely local, autonomous transactions which are under the control of the local DBMS and ensure that the consistency of the database is maintained. It is impossible to synchronize local and global transactions while still preserving local autonomy [Bel92]. Once the global transaction submits a subtransaction to the local DBMS, it effectively relinquishes control over it. The local DBMS will assume all responsibility and will decide whether to commit or reject and roll-back the transaction. Hence, some local DBMS could commit and others could abort the same transaction, thereby destroying the atomicity of the global transaction and compromising the consistency of the multidatabase system.

The traditional approach to ensuring that histories preserve database consistency in a multidatabase requires histories to be serializable. In a MDMS environment, a history is serializable if and only if global transactions in the history are serialized in the same order in all local database systems [Bre88]. However, even serial execution of global transactions does not guarantee global serializability.

The problems of the provision of general support for global transactions in the presence of local transactions are as follows:

- Maintaining global transaction atomicity.
- Global serialization.
- Detection and prevention of global deadlock.

Most existing multidatabase systems support retrieval only — all updates must be done locally. Even with this restriction, the problem of dirty or unrepeatable reads must be addressed. This means that read-only transactions which require a consistent view of the database have to take into account that the database could be simultaneously updated by local transactions.

4.3.1 The global transaction atomicity problem

If local sites want to preserve their execution autonomy, then they probably will not support a prepare-to-commit command. In this case, a DBMS can unilaterally abort a subtransaction at any time before its commit. This leads to global transactions that are not atomic and incorrect global schedules as well.

Example 4.2 *: Global transaction atomicity problem*

Lets go back to our pharmacy example in Figure 1.1. Using the transaction model introduced in Chapter 3 and extended here, let us refer to the Tonic pharmacy as LDB^1, the Medilots pharmacy as LDB^2 and the Harbour pharmacy as LDB^3. Assume the data items at each local database are as follows:

$$\mathcal{LDB}^1 = \{d, e, f, g\}$$

$$\mathcal{LDB}^2 = \{s, t, u, v\}$$

$$\mathcal{LDB}^3 = \{w, x, y, z\}$$

Consider the following global transaction: GT_1:

$$GT_1 \ : \ r_1(d); \ \ w_1(d); \ \ w_1(s); \ \ c_1;$$

Suppose that GT_1 has completed its read/write actions at both sites and the GTM sends commit requests to both sites. LDB^2 receives the commit and commits its subtransaction. However, LDB^1 decides to abort its subtransaction before the commit arrives. Therefore, at LDB^1 the local DBMS undoes GT_1's actions. After

this is accomplished, a local transaction L at LDB^1:

$L : r_2(d);\ w_2(d);\ c_2;$

is executed and committed at the site. At this point, the resulting global history is incorrect as it only reflects the LDB^2 half of GT_1. To correct the situation, the GTM may attempt to send the missing write $w_1(d)$. This is referred to as a redo of the transaction. The local DBMS will interpret this as a new transaction GT_2 which is not related to GT_1. Thus what has transpired is:

$r_2(d);\ w_2(d);\ w_3(d);$

However, GT_2's write operation is the same as $w_1(d)$ as far as the MDMS is concerned and what has actually transpired is the following non-serializable history:

$r_1(d);\ r_2(d);\ w_2(d);\ w_1(d);$

◇

If the DBMS provided a prepare-to-commit operation and participated in a global commit protocol, then the problems shown in the example above could be avoided. In the above example, the GTM would not issue the commit transactions for GT_1 until both sites had acknowledged the prepare-to-commit. Because LDB^1 is prepared for GT_1, it cannot abort it and the situation described above does not arise.

There is an ongoing debate about whether sites in a MDMS should be required to provide prepare-to-commit operations and give up their execution autonomy. While some argue that 2PC is standard and should be provided, others argue that there will always be autonomous sites that want to preserve autonomy and do not want to provide this command. They do not want their sites to hold locks for remote sites which could then be held for an indefinite period of time. The proponents of 2PC argue that because networks are very fast, the period of time that a lock will be held is minimal and anyway, if there is a protracted wait, an operator can break it, but the other camp now reiterate that in this case we are back to a state where unilateral aborts can take place anyway so are back to square one [Bre95]. The problem of transaction atomicity forms an integral part of the reliability of a multidatabase system. It is discussed in Chapter 5.

4.3.2 The global serialization problem

Ensuring serializability in a MDMS environment is a difficult task [Ras93b]. This difficulty is exacerbated by the design autonomy of the local database systems and the fact that the local systems are pre-existing, which implies that they may follow different concurrency control algorithms. The various local DBMSs integrated by the MDMS may use different

concurrency protocols eg. *two-phase-locking* ($2PL$), *timestamp ordering* (TO), *serialization graph testing* (SGT) etc. Hence existing solutions for homogenous distributed database systems cannot be used in an MDMS environment. Furthermore, since the local transactions execute outside the control of the GTM, the resulting execution may not be serializable.

Example 4.3 *:* The global serialization problem

Consider once again the multidatabase in Example 4.2. Say we have the following two non-conflicting transactions:

$GT_1 : r_1(d); r_1(s);$

$GT_2 : r_2(e); r_1(t);$

In addition, consider the following local transactions; L_3 executing at LDB^1 and L_4 executing at LDB^2.

$L_3 : w_3(d); w_3(e);$

$L_4 : w_4(s); w_4(t);$

Now consider a history in which transaction GT_1 first executes at sites LDB^1 and LDB^2 followed by the execution of transaction GT_2 at both LDB^1 and LDB^2. It is possible for the local transactions L_3 and L_4 to execute in such a manner that GT_1 is serialized before GT_2 at LDB^1, while GT_2 is serialized before GT_1 at LDB^2. For example:

$LH^1 : r_1(d);\ c_1;\ w_3(d);\ w_3(e);\ c_3;\ r_2(e);\ c_2;$
$LH^2 : w_4(s);\ r_1(s);\ c_1;\ r_2(t);\ c_2;\ w_4(t);\ c_4;$

As far as the GTM is concerned, global transactions GT_1 and GT_2 are executed serially. At LDB^1, the resulting execution is serial: GT_1, L_3 and GT_2 . At LDB^2, the resulting execution is also serial: GT_2, L_4 and GT_1 .

Yet, if we look at the global execution, it is non-serializable because, to be serializable, GT_1 should always precede GT_2 or vice versa. ◇

This problem arises because the local transactions create *indirect* conflicts between global transactions. Since the GTM is not aware of local transactions, it is also not aware of these conflicts. This is the cause of major difficulties in a multidatabase environment [Bre95].

In this example, since the GTM is unaware of the indirect conflicts between global transactions at the local database systems due to the execution of local transactions, the resulting global history is non-serializable. The local DBMS will also not communicate any information relevant for concurrency control to the GTM because of nodal autonomy. Because of this, the GTM has no idea of the serialization order of transactions at local database systems. Thus, to ensure serializability, the GTM would need to assume an indirect conflict between global transactions even though in reality they do not conflict at the local database systems. Hence, adopting serializability as the correctness criteria in MDMS environments could result in a low degree of concurrency and poor performance [Ras93b].

Because of this, most prototype systems built only allow retrieval of data by global transactions and do not have any concurrency control schemes to co-ordinate the execution of global transactions. This makes it possible for global queries to retrieve inconsistent data in these systems. The research into this problem is discussed in section 4.4.

4.3.3 The global deadlock problem

In MDMS systems there is a possibility of global deadlock that cannot be detected by the GTM. We can illustrate the problem in the following example:

Example 4.4 *:* The global deadlock problem

Consider our example multidatabase in Example 4.2 again. Local DBMSs at both sites use the two-phase locking protocols to guarantee local serializability. Let GT_1 and GT_2 be two global transactions defined as follows:

$GT_1 : r_1(d);\ r_1(t);$
$GT_2 : r_2(s);\ r_2(e);$

In addition, let L_3 and L_4 be two local transactions at sites LDB^1 and LDB^2 respectively, defined as follows:

$L_3 : w_3(e);\ w_3(d);$
$L_4 : w_4(t);\ w_4(s);$

Assume that GT_1 has executed $r_1(d)$ and GT_2 has executed $r_2(s)$. After that,

at LDB^1 local transaction L_3 executes $w_3(e)$, submits $w_3(d)$ and is forced to wait for a lock on d that is kept by GT_1. At LDB^2 local transaction L_4 executes $w_4(t)$, submits $w_4(s)$ and is forced to wait for a lock on s that is kept by GT_2. Finally, transactions GT_1 and GT_2 submit their last operations and a global deadlock ensues. ◇

Due to design autonomy, local DBMSs may not wish to exchange control information and will therefore be unaware of the global deadlock. The MDMS is unaware of the local transactions and is therefore also unaware of the deadlock. The research into this problem is discussed in section 4.5.

4.4 Global Concurrency Control

Concurrency control issues in multidatabase systems were first discussed by Gligor and Popescu-Zeletin [Gli86]. They outlined the basic requirements for a transaction management scheme to ensure database consistency in an MDMS environment, and pointed out difficulties related to transaction management in such systems. Since then, a number of schemes addressing the concurrency control problems in multidatabases have been proposed. Most of the existing schemes preserve database consistency by ensuring that global schedules are serializable. Non-serializable schemes have also been presented where the serializability requirement is relaxed. Other schemes enhance the degree of concurrency while still ensuring serializability of schedules by, for example, exploiting the semantics of operations or relaxing the atomicity requirement by resorting to compensation. [Ras93b] and [Meh93] have studied and evaluated various concurrency control schemes which have been proposed for handling concurrent transactions in multidatabases. An overview of their comments is given here as well as an evaluation of other schemes not presented by them.

4.4.1 Serializable executions

The great advantage of serializability is the simplicity thereof. The application programmer does not have to worry about the correctness of concurrent executions. The programmer only has to worry about the consistency of the database being maintained. Protocols for ensuring serializability are simple, easily implementable and can be followed by the transaction manager to ensure that schedules are serializable. [Ras93b]

However, adopting serializability as a correctness criteria could adversely affect the performance of the system. In some systems, a weaker notion of correctness is desirable.

Proposed serializability schemes outlined in this section are either *pessimistic* or *optimistic.*

- In pessimistic schemes, the GTM does not submit a global transaction operation to the local DBMSs if its execution could potentially lead to a non-serializable schedule.

As a result, pessimistic schemes result in very few aborts, but permit a low degree of concurrency.

- The optimistic schemes, on the other hand, the GTM permits transaction operations to execute freely, but commits a global transaction only if its commitment cannot result in non-serializable schedules. These schemes could result in low performance because of frequent global transaction aborts.

Schemes that ensure serializability can be further classified according to whether or not they preserve the local autonomy of sites. [Ras93b]

4.4.1.1 Schemes that preserve local autonomy

The schemes discussed in this section do not violate design, communication and execution autonomies of the local database systems. Local transactions are assumed to execute outside the control of the GTM and every global transaction is assumed to have only one subtransaction executing at any local site.

Synopsis 4.1 $\Longrightarrow$ Gligor *et al*'s altruistic locking scheme

The first concurrency control schemes were presented by [Gli86] and [Alo87] and were pessimistic. In [Gli86], Gligor and Popescu-Zeletin propose a scheme in which the GTM determines serialization orders globally and then enforces them locally at the local DBMSs. The scheme in [Alo87] uses an *altruistic locking protocol* [Bre92d] for controlling the submission and execution order of global transactions. The altruistic locking protocol is based upon locking the site at which a global subtransaction executes. In both schemes there may still be indirect conflicts between global transactions due to the execution of local transactions at local database systems and this could violate global serializability. [Ras93b]

Synopsis 4.2 $\Longrightarrow$ Breitbart *et al*'s site graph scheme

[Bre88] presents a pessimistic scheme for ensuring serializability in a MDMS environment based on the notion of a *site graph*. This is one of the first schemes developed that correctly ensures global serializability. A site graph is an undirected bi-partite graph consisting of nodes corresponding to local sites (site nodes) and global transactions (transaction nodes). When a transaction begins execution, edges are inserted into the site graph between the node corresponding to the transaction and the sites at which the transaction executes. The transaction will be aborted if insertion of the edges causes a cycle in the site graph. This scheme *does* ensure that consistency of the database is maintained, but only provides a low degree of concurrency because two or more global transactions cannot execute if they have more than one site in common. Further, based only on information on the order in which global subtransactions execute at the local DBMSs, it may not be possible to delete edges from the site-graph without potentially risking loss of serializability.

Breitbart *et al* [Bre90a] later worked on a scheme which required global transactions to obtain locks on local data items at the GTM and also permitted multiple transactions to be in the prepare to commit state at any site.

Synopsis 4.3 $\Longrightarrow$ Elmagarmid *et al*'s serialization event scheme

Elmagarmid and Du [Elm90a] state that in a number of concurrency control strategies, the serialization order of a transaction depends on the execution of an event, referred to as the *serialization event.* For example, for the timestamp ordering concurrency control protocol, the serialization event for a transaction is the operation that results in the transaction being assigned a timestamp. Similarly, in the 2PL concurrency control protocol, the serialization event for a transaction is the operation that results in the transaction obtaining its last lock. By ensuring that the serialization events of global transaction are executed in the same order at all the DBMSs, the GTM can ensure that global schedules are serializable. The notion of serialization events is also used in [Meh92a] where they reduce the serializability problem in multidatabases is reduced to the problem in a centralized database. Serialization events can also be used to alleviate some of the problems associated with the site-graph scheme which was proposed in [Bre88].

Synopsis 4.4 $\Longrightarrow$ Wolski's 2PC agent method

In [Wol90], the authors propose a solution called the 2PC agent method, which assumes that the participating local DBMSs use 2PL and produce only strict schedules. This scheme does not ensure global serializability since local strictness is not sufficient in order to ensure serializability of schedules in a multidatabase environment. The 2PC agent method is extended to an environment where local DBMSs produce only rigorous schedules in [Vei92]. The proposed method is totally decentralized, and requires local transactions not to update data accessed by global transactions in the prepared to commit state. Unlike the scheme in [Bre90a], which requires global transactions to obtain locks at the GTM and permits multiple transactions to be in the prepared state at a site, the scheme in [Vei92] permits only one global transaction per site to be in a prepared state at any given time, and requires commit operations belonging to global transactions to be submitted to local DBMSs in a globally unique total order. [Ras93b]

This scheme also requires local transactions not to update certain data items.

Synopsis 4.5 $\Longrightarrow$ Georgakopoulos's optimistic ticket method

Georgakopoulos [Geo90] defines the concept of *rigorous schedules* and schedulers. He then proves that by using such schedules one can ensure that serialization and execution orders are analogous.

If one has underlying local database systems which do not produce rigorous schedules, they propose another scheme to avoid inconsistent retrievals. He introduces the *optimistic ticket method* (OTM). OTM is a multidatabase transaction management mechanism that guarantees global serializability by permitting execution of multidatabase transactions only when their relative serialization order is the same in all participating LDBs. OTM requires the LDBs to guarantee only local serializability.

To assure correctness if the schedulers do produce rigorous schedules he introduces the *implicit ticket method* (ITM) which is a refinement of OTM that eliminates ticket conflicts, but works only when participating LDBs use rigorous transaction scheduling mechanisms. Contrary to Wolski's 2PC agent method, this method does not assume that the local schedulers use 2PC locking and also does not require local sites to provide a prepare-to-commit state [Geo90]. The Stargate prototype [Key93] uses this method.

Synopsis 4.6 $\Longrightarrow$ Georgakopoulos *et al*'s forced conflict scheme

Some concurrency control protocols do not have easily identifiable serialization events (e.g. SGT). Serialization events can be introduced for these protocols by some external means by forcing conflicts between transactions [Geo91b].

In [Geo91b], Georgakopoulos, Rusinkiewicz and Sheth present an optimistic MDMS transaction management mechanism that permits the commitment of global transactions only if their serialization order is the same at all participating local DBMSs. The basic idea in the scheme is to create direct conflicts between global transactions at each local DBMS that allows the GTM to determine the relative serialization order of their subtransaction at each site. Every global subtransaction at a site is forced to read a data item, eg *ticket*, and then increment it by one. The value of *ticket* is used to determine the relative serialization order of the subtransaction at the site. The scheme requires the local DBMS to guarantee only local serializability. [Ras93b]

Synopsis 4.7 $\Longrightarrow$ Batra *et al*'s decentralized GTM scheme

The idea of forcing conflicts is also used in [Bat92] to develop a pessimistic, fully decentralized, deadlock-free global concurrency control method. Each global transaction is assigned a system-wide unique timestamp locally at the sites at which it is submitted. The GTM at every site then ensures that global transactions are serialized at a particular site in the order of their timestamps by requiring transactions to write on ticket, in timestamp order, the value of their timestamps. A drawback with this scheme is that if the sites are far apart,

then there could be a large number of aborts (timestamp values must be kept approximately synchronized between the various sites for better performance). [Ras93b]

Synopsis 4.8 $\Longrightarrow$ Breitbart *et al*'s rigorous schedule scheme

In [Bre91a], the authors introduce the notion of *rigorous schedules* which are schedules that, in addition to being strict, have the property that no transaction writes a data item until the transaction that previously read it either commits or aborts. A local DBMS produces rigorous schedules if it delays the execution of an operation O_i (belonging to transaction T_i) in case it has previously scheduled an operation O_j (belonging to transaction T_j) that conflicts with O_i, until the commitment of transaction T_j. A number of concurrency control algorithms currently produce rigorous schedules (eg strict 2PL). If all the local schedules are rigorous and the commit operation of a global transaction is submitted only after the transaction has completed its execution at all DBMSs, then the serializability of global schedules is ensured (in the absence of failures).

An additional observation made in [Bre91a] is that if in each of the participating DBMSs the serialization order of transactions is the same as their commitment order, then the GTM can ensure global serializability by controlling the order of global transaction commits. Based on this observation, they propose an additional class of schedules, namely the *strongly recoverable*[3] schedules, in which the serialization event for a transaction is its commit operation. An algorithm that ensures global serializability in failure prone MDMS environments in which each local DBMS generates only strongly recoverable schedules is developed in [Bre92a].

Synopsis 4.9 $\Longrightarrow$ Raz's commitment ordering

The *commitment ordering* (CO) property introduced by [Raz92] is the same as strong recoverability introduced by [Sop91b] (see synopses 4.17, 4.8) and defined in definition 3.19. In [Raz92], the author proposes various blocking as well as non-blocking implementations of CO for local database systems. Raz also examines the relationship between properties of schedules generated by local DBMSs and properties of global schedules. For example, if every local DBMS generates strict schedules, then global schedules are also strict. This does not hold for serializability [Ras93b]. The author shows that CO of local schedules is both a necessary and sufficient condition for guaranteeing global serializability in an environment consisting of autonomous local database systems. [Ras93b]

[3] See definition 3.19

Synopsis 4.10 $\Longrightarrow$ Breitbart *et al*'s partitioning scheme

All of the above-mentioned schemes (with the exception of [Bre92a]) assume that a local DBMS cannot unilaterally abort a transaction at any point during its execution. In [Bre90a, Bre92b, Meh92c], transaction management schemes that preserve the execution autonomy of local DBMSs as well as ensure atomicity and serializability of transactions in a failure prone MDMS environment are presented. The GTM maintains global locks for data items accessed by global transactions and imposes restrictions on data items accessed by local and global transactions.

They also propose a scheme for detecting deadlocks in an MDMS environment. This scheme requires that local DBMSs use the strict 2PL concurrency control protocol. To deal with system failures and transaction aborts, the authors introduce the redo approach to recovery. In their scheme, a 2PC protocol is used in which the MDMS software and the servers, rather than the local DBMSs, participate in the protocol in order to commit global transactions.

Since the local DBMSs have complete control over transactions at their site, they may abort a subtransaction of a global transaction even though it is considered committed by the MDMS software. The MDMS maintains logs in order to facilitate the redo-ing of aborted global subtransactions. Since the MDMS software has no control over the execution of local transactions, redoing the writes of aborted subtransactions may result in a loss of database consistency. This is dealt with by partitioning the set of data items at a local DBMS into:

- globally updateable — those data items that can be updated only by global transactions, and
- locally updateable — those data items that can only be modified by local transactions.

Further, a global transaction cannot read any locally updateable data item if it modifies the values of any global data item.

This scheme was extended in [Meh92c] to cases where the local DBMSs perhaps do not use the strict 2PL protocol. [Meh92c] also introduced the concept of *semi-rigorous* schedules[4]. The scheme proposed in [Bre92b] also employs a structure similar to the site-graph, called the *commit graph*, in order to coordinate the commitment of global transactions at the local database sites. The authors also address the issue of global deadlock detection [Meh93].

A major drawback of these schemes is that they require local transactions not to update certain data items [Ras93b, Meh93].

[4] See definition 3.20

Synopsis 4.11 $\Longrightarrow$ Kang & Keefe's distributed strict timestamp ordering scheme

Kang & Keefe [Kan93] propose a *distributed strict timestamp ordering scheme* (DSTO) which is globally serializable. Kang & Keefe also define the notion of one-copy serializability as a correctness criteria in the place of conflict serializability.

> A global history is *one-copy serial* if for all i, j and x, if T_i reads x from T_j in GH, then $i = j$ or T_j is the last transaction preceding T_i that writes into any version of x in GH.
>
> Two global histories are equivalent over a set of transactions if they have the same operations.
>
> A global history is *one-copy serializable* if it is equivalent to a one-copy serial history.

In DSTO, each global transaction is assigned a unique global timestamp when it starts. Each subtransaction carries the parent's timestamp. The GTM at each site executes strict timestamp ordering. Strict TO blocks transactions attempting to read or write an object until the transaction that previously write it has either committed or aborted. The GTM ensures that conflicting operations are executed at the local site in global timestamp order by aborting transactions whose operations arrive too late.

All global subtransactions are required to take-a-ticket (ticket being an object not updateable by local transactions). We only assume that the local data manager at each site outputs serializable and cascadeless schedules. Kang & Keefe also require the objects to be partitioned into locally and globally updateable sets.

Kang & Keefe prove that the DSTO scheme produces globally serializable histories in the face of failures and also prove that the scheme is deadlock free [Kan93].

Synopsis 4.12 $\Longrightarrow$ Mehrotra *et al*'s serialization function scheme

[Meh92a] reduces the problem of ensuring global serializability in a multidatabase environment to that of ensuring it in a centralized database system. Concurrency control in centralized database systems has been well studied and this therefore makes the concurrency control problem in multidatabases more manageable.

[Meh92a] introduces the notion of serialization functions in order to assist in serializing transactions in a multidatabase. [Meh92a] has presented a number of conservative concurrency control schemes to be used in conjunction with the serialization functions in order to ensure global serializability. These schemes still need to be made fault-tolerant.

Synopsis 4.13 $\Longrightarrow$ Yun *et al*'s PTM scheme

Yun & Hwang [Yun93] propose a new concurrency control algorithm called the *pessimistic timestamp method* (PTM). PTM approaches the concurrency control problem the following way: the global scheduler does not schedule any operation of the global transaction which has the possibility of violation of global serializability.

PTM schedules the global transactions so that the serialization order of global transactions is the same as their execution order at all participating LDBs. PTM also disallows the abort of a global transaction that is being executed due to the violation of global serializability or the occurrence of global deadlock except for transaction and site failures. Finally, PTM assigns a timestamp to a local transaction, or a global subtransaction, when it is scheduled (or the subtransaction arrives at its prepare-to-commit state). The timestamps are used to resolve the discrepancy between the execution order and the serialization order of global transactions.

PTM preserves local autonomy, achieves global serializability, and achieves a high degree of concurrency. There is also no deadlock problem when using this algorithm.

A disadvantage of this approach is that there will be a high overhead for cycle detection and the storage space needed to keep all details of timestamps would also constitute a high overhead.

4.4.1.2 Violation of local autonomy

We can now consider the algorithms which violate local autonomy in order to ensure serializability of global schedules.

Synopsis 4.14 $\Longrightarrow$ Zhang *et al*'s hybrid approach

Zhang and Orlowska [Zha93] have proposed a hybrid concurrency control approach which is a combination of an optimistic concurrency control mechanism and an optimistic ticket method. Their algorithm assumes that local DBMSs are able to distinguish between local transactions and subtransactions of global transactions, and that local DBMSs will provide ready-to-commit information to the GTM. They also assume that local database systems produce strict schedules and resolve local deadlocks. This scheme sacrifices local autonomy in order to improve multidatabase performance.

Synopsis 4.15 $\Longrightarrow$ Pu's DBMS modification approach

A scheme which violates local autonomy is presented in [Pu88]. The authors assume that each local DBMS can be modified to return the serialization order of each global transaction executed at the local site to the GTM. The GTM then uses the serialization orders from all the local sites to validate the execution of a global transaction. The approach provides a high level of concurrency [Ras93b].

Synopsis 4.16 $\Longrightarrow$ Perrizo *et al*'s pessimistic protocol

In [Per91], global serializability is ensured by routing local transactions through a local server module prior to being submitted to the local DBMS. This is achieved by giving the server the same name, feel and look as the DBMS. They implement a pessimistic concurrency control protocol in which the GTM determines what the ordering of the global transactions will be. The local server uses a scheme similar to the SGT concurrency control scheme to ensure that global transactions are serialized. Of course, the local DBMSs lose all control over their databases and this scheme also results in poor performance due to the duplication of the locking mechanisms in all the local database systems (i.e. at the local server as well as in the local DBMS) [Ras93b].

Synopsis 4.17 $\Longrightarrow$ Soparkar *et al*'s violation of autonomy scheme

In [Sop91a], the authors present a pessimistic scheme that requires local database systems to give up control of their databases. This scheme requires local transactions to execute under the control of the GTM. The local database systems can follow any concurrency control protocol as long as every transaction performs a read before every write and avoids cascading aborts (ACA). The scheme reduces overhead by minimizing the GTM control over execution of transactions for the purpose of ensuring serializability. Serializability is ensured by forcing conflicts between global subtransactions at the local database system. The authors exploit the ACA property of local schedules, the fact that writes are preceded by reads, and the two-phase-commit protocol in order to ensure that global transactions are serialized at local sites. This scheme does not ensure serializability in the presence of failures. [Ras93b]

In [Sop91b] the authors replace the ACA requirement on local databases by *strong recoverability* and adopt a scheme where the global atomic protocol is used to coordinate commit operations belonging to global subtransactions so that global serializability is assured. If a committed global transaction is aborted by some local DBMS, the set of active transactions that potentially conflict with T_i at the local DBMS will be aborted and T_i's writes are re-submitted to the local DBMS.

4.4.2 Relaxing serializability

Abandoning serializability as a correctness criteria could complicate concurrency control from the point of view of the programmer as well as the GTM. Serializability has been shown to be a sufficient but not necessary requirement for ensuring that concurrent execution of transactions preserve database consistency [Ras93b].

Most of the schemes that use serializability cause a low degree of concurrency and usually perform poorly. Some researchers have proposed that the serializability requirement be relaxed and that alternative correctness criteria be investigated for multidatabases.

In this section, various schemes for ensuring correctness without enforcing serializability are discussed.

4.4.2.1 Schemes that exploit knowledge of integrity constraints

Synopsis 4.18 $\Longrightarrow$ Du and Elmagarmid's quasi-serializability

In [Du89], the authors introduce the notion of *quasi serializability* (QSR).

- A global schedule S is *quasi serial* iff local schedules are serializable and there is a total order on global transactions such that, for any two global transactions T_i and T_j in S, if T_i precedes T_j in the total order, then all of T_i's operations precede all of T_j's operations at each and every local site.
- A global schedule is QSR if it is conflict equivalent to a quasi serial schedule.

The authors claim that if QSR schedules are to preserve database consistency, and QSR is to be used as a correctness criteria in an MDMS environment, then the following must hold:

1. There must be no integrity constraints between data items at different sites except those arising from replication.
2. Global transactions must not have *value dependencies* - that is, the execution of a global transaction at a site must be independent of its execution at other sites.
3. Local transactions must not be permitted to write replicated data.

A pessimistic deadlock-free algorithm that does not violate the local autonomy of sites and ensures that schedules in a MDMS are QSR is presented in [Vei92]. A data structure similar to a site graph — called an *access graph* — is maintained and execution of a global transaction is delayed if insertion of its edges causes a cycle. [Ras93b]

Synopsis 4.19 $\Longrightarrow$ Rastogi's 2LSR scheme

Rastogi [Ras93b] proposes two approaches for relaxing the serializability requirement. These approaches highlight the trade-off between the extent to which users are shielded from the formidable task of proving the correctness of non-serializable executions, and the performance improvement obtained as a result of exploiting the semantics of operations. [Ras93b] proposes a new correctness criterion for MDMS environments — *two-level serializability* (2LSR). A schedule is 2LSR if

- each of the individual DBMSs generates serializable schedules, and
- the restriction of the schedule to only global transactions is serializable.

Any protocols for ensuring serializability in centralized DBMSs can be adapted to ensure 2LSR in an MDMS environment. Rastogi further proves that 2LSR schedules preserve database consistency in certain MDMS models based on partitioning of data items at each site, and restricting the read and write operations of the various data items. The problem with this scheme is that assumes schedules to only consist of read and write operations. On the other hand, it does free users from the task of proving correctness of non-serializable executions.

Synopsis 4.20 $\Longrightarrow$ Korth *et al*'s predicate-wise serializability

Korth, Kim and Bancilhon [Kor88] propose the notion of predicate-wise serializability (PWSR). If database consistency is expressed as a conjunction of predicates, then the restriction of a PWSR schedule to the set of data items in every conjunct is serializable. In [Ras93b], it is shown that PWSR schedules preserve database consistency if transactions and integrity constraints are of a restricted nature.

Synopsis 4.21 $\Longrightarrow$ Mehrotra *et al*'s RS-correctness scheme

Mehrotra [Meh92d] proposes a new model of correctness called RS-correct schedules. This scheme exploits the integrity constraints of the system to produce non-serializable schedules. They identify two types of constraints, implicit and explicit. Explicit constraints are easily defined — they usually would be something like "the balance of an account must always be positive", but implicit constraints are difficult to define by just expressing them via the data items themselves. An implicit constraint might be something like: "a transaction must always see the correct balance when accessing customer accounts". To deal with this, Mehrotra defines RS-correctness by firstly defining two types of global transactions: RS-transactions which need to see database states consistent with both implicit and explicit constraints, and non RS-transactions which are required to see database states only consistent with respect to explicit constraints. Now a schedule can be defined to be RS-correct if:

- It preserves the explicit integrity constraints of the database,
- Transactions in S see database states consistent with respect to explicit integrity constraints, and
- No cycle in the serialization graph of S contains an RS-transaction (that is, no transaction in S is serialized both before and after an RS-transaction).

These schedules preserve integrity constraints of the database and also ensure that transactions see the correct database states. Their protocol combines the 2PL protocol and the scheme of forcing local conflicts between transactions as presented by [Geo91b]. In [Meh92d], the authors call this new protocol *Forced Conflict 2PL (FC2PL)*. They then

prove that if the FC2PL protocol is used, every global schedule is serializable. It does not infringe on the local autonomy of the various sites. The only problem with this scheme is that it is not very fault tolerant because the locking protocol could cause a deadlock situation in the case of a site failure.

Synopsis 4.22 $\Longrightarrow$ Barker's M-Serializability

Barker [Bar90] proposes a new correctness criterion called *M-serializability* which is an extension of serializability theory. M-serializable histories form a superset of serializable histories. His theory captures the characteristics of both local and global transactions. Further, multidatabase serialization graphs are developed to make it easy to determine when a multidatabase history is M-serializable.

Barker's method does not violate local autonomy. The problems with this approach are that the two-phase commit in his model does not permit value-dependencies that span local database system boundaries. This means that values at multiple database systems cannot be checked. Secondly, this approach requires that all DBMSs provide a strict level of service [Bar90].

Synopsis 4.23 $\Longrightarrow$ Jin *et al*'s FT-Serializability scheme

Jin *et al* [Jin93] propose the notion that the serialization order of flexible transactions (see 'Rusinkiewicz's flexible transactions' in section 4.2.9.2) should be the same only at sites where they conflict. This scheme is applicable to the particular environment of service provisioning (the activity of setting up a telecommunication service based on a customer's requests). A flexible transaction (FT) is specified by providing the following: the precondition of the global transaction, a set of subtransactions, externally visible states of each subtransaction, possible transitions among these externally visible states, pre- and postconditions for the possible transitions of each subtransaction, the postcondition of the global transaction.

They define FT-serializability as follows:

> A global history is FT-serializable if for any subtransactions GST_i^x and GST_j^x $\in FT_x$ and GST_i^y and $GST_j^y \in FT_y$ such that GST_i^x conflicts with GST_i^y and GST_j^x conflicts with GST_j^y then $GST_i^x \prec GST_i^y \Rightarrow GST_i^x \prec GST_i^y$ at all sites where they conflict.

The authors rely on the concurrency control mechanisms of the local systems to ensure that subtransaction submitted to local systems will be executed correctly with respect to local concurrency control. Therefore, the lock held by a subtransaction can be released as soon as the subtransaction completes its submission phase. This algorithm allows a higher degree of concurrency than the altruistic locking algorithm introduced by [Alo87] although it uses the same locking granularity.

The mechanism introduced here is less general than other proposed solutions but allows a higher performance in the specific real world environment in which it is applied.

Discussion

The above approaches relax the serializability requirement and can be shown to preserve the integrity constraints of the database, but they do not address the issue of whether or not the preservation of integrity constraints, by itself, is a sufficient consistency guarantee for transactions. The answer to this depends on the particular application. Examples of where this is not true can be seen in [Meh93].

4.4.2.2 Schemes that exploit transaction semantics

This section discusses schemes that exploit the semantics of transactions to relax the serializability requirement. Schemes that exploit transaction semantics consider each transaction to consist of a number of subtransactions with each of them having a type associated with it. The application administrator specifies the various subtransaction types and also the various interleavings of the subtransactions that will not result in a loss of database consistency. A transaction manager will utilize this specification to permit only acceptable, and prevent unacceptable interleavings of the transactions.

Each of the schemes that exploit transaction semantics are based on specifying the acceptable/unacceptable interleavings to the transaction manager. They differ only in the mechanism they employ.

Synopsis 4.24 $\Longrightarrow$ Chen *et al*'s distributed GTM scheme

Many of the schemes above guarantee consistency of the database only if specified conditions are satisfied [Alo87, Bre88, Geo91b, Bat92, Meh92a]. It would therefore be helpful if the MDMS administrators could utilize the semantics of global transactions and the concurrency control strategies of the underlying local DBMSs to customize a global concurrency control approach. Just such a scheme is presented in [Che93]. The architecture for Chen *et al*'s transaction processing model was discussed in synopsis 4.2.5.

This algorithm combines *two-phase locking* and the *linear ordering of resources*. By doing this, the algorithm provides a deadlock-free, totally distributed, and correct synchronization of concurrent scheduling order requests from global transactions. The typical global transaction will be performed in two phases. In the first phase, the relative scheduling order of a global transaction with respect to other global transactions is determined. This means that the algorithm applies a type of two-phase-locking for each transaction where all MDMS interfaces must be locked at every LDB at which a subtransaction of that transaction must be run. After this locking has been done, the scheduling order is determined after which the interfaces are unlocked and the transaction (ie all its subtransactions) is executed in the second phase.

The advantages of this algorithm are that the concurrency control decisions concerning a global transactions are made independently either by the GTM or by the individual MDMS interfaces in the LDBs. This algorithm is therefore fully decentralized and the GTM is distributed among all the machines where global transactions can be issued. [Che93] contends that their approach is more flexible and reliable than the algorithms presented in [Meh92a, Alo87, Vid91, Bre88, Du89].

The disadvantages of this approach are that performance is lowered by additional network delays caused by the additional network traffic. This scheme also reduces concurrency compared to other algorithms. The network delays can be alleviated by high speed networks and the reduced concurrency is offset by the fact that no global transactions will be aborted due to deadlocks or nonserializable executions [Buk93].

Synopsis 4.25 $\Longrightarrow$ Garcia-Molina & Salem's saga scheme

Another approach is the *saga* which is specified in [Gar87]. In this transaction model, a transaction is broken into a sequence of subtransactions each of which is an independent activity by itself. In the saga model all possible interleavings are permitted. If every global transaction is a saga, none of whose subtransactions execute at more than one site, then since local schedules are serializable, no global concurrency scheme is required.

Synopsis 4.26 $\Longrightarrow$ Lynch's and Garcia-Molina's compatibility set schemes

The saga approach may not be very effective in database environments where certain interleavings between steps are undesirable. In order to remedy the problem, the schemes in [Lyn83, Gar83] associate types with transactions, and mechanisms that use the type information for specifying acceptable interleavings between steps are developed. The authors also develop protocols for ensuring that only the specified interleavings are permitted.

In [Gar83], the set of permissible interleavings of subtransactions are specified by grouping transactions into *compatibility sets*. Steps of transactions whose types belong to a single compatibility set are permitted to interleave freely, while steps of transactions belonging to distinct compatibility sets are not permitted to interleave at all. A locking protocol is used to prevent undesirable interleavings. The concept of compatibility is discussed by [Lyn83] and several levels of compatibility among transactions are defined. These levels are structured hierarchically so that interleavings at higher levels include those at lower levels. Further, [Lyn83] introduces the concept of breakpoints within transactions which represent points at which other transactions can interleave. Similar ideas have been proposed in [Vei89]. Note that it is the responsibility of the user to specify interleavings that will maintain the database integrity constraints [Ras93b].

Synopsis 4.27 $\Longrightarrow$ Rastogi's graph based approach

Rastogi [Ras93b] cites two approaches. One is the 2LSR approach (see Synopsis 4.19) while the other exploits semantics of operations. In this approach, the set of undesirable interleavings are specified as regular expressions over the types of subtransactions in the system. The expressions used here are more general than compatibility sets since using regular expressions allows us to specify certain interleavings which cannot be specified using compatibility sets. [Ras93b] also develops an algorithm that the MDMS can use to prevent unacceptable interleavings that are specified as regular expressions. The algorithms are graph-based and involve searching for cycles in the graph that satisfy certain properties. This scheme also allows a higher degree of concurrency than schemes which ensure global serializability.

4.4.2.3 Schemes that tolerate bounded inconsistencies

Schemes which are discussed in this section tolerate a certain degree of inconsistency as long as the degree of inconsistency is bounded. In some systems it is not that important to use exact values (e.g. statistical information gathering). The schemes mentioned in this section will attempt to quantify the degree of the inconsistency.

Synopsis 4.28 $\Longrightarrow$ Pu & Leff's epsilon serializability scheme

Pu & Leff [Pu91a] develop one such approach. The authors propose a notion of *epsilon-serializability* (ESR).

> A schedule is ESR if the restriction of the schedule to only update transactions is serializable, and the inconsistency associated with every transaction is less that the amount specified for it.

A divergence control mechanism for ensuring schedules are ESR, based on the 2PL protocol, is proposed in [Wu92]. Transactions are classified into read-only or update transactions. The projection of the schedule to operations belonging to update transactions is required to be serializable and thus the consistency of the database is preserved. However, the schedule itself may not be serializable and queries may retrieve inconsistent data. The degree of inconsistency is measured by counting the number of conflicts it is involved in, which if not present, would make the schedule serializable. With each query, we associate a maximum number of conflicts that can be allowed. Non-serializable schedules are permitted if the number of conflicts does not exceed this maximum amount [Meh93].

Synopsis 4.29 $\Longrightarrow$ Wong & Agrawal's seralizability with bounded inconsistency

In [Won92], a similar approach is developed in the context of an object based database sysem in which the authors propose a notion of *serializability with bounded inconsistency.* In their approach, each operation is associated a maximum level of inconsistency that can

be permitted. A schedule is serializable with bounded inconsistency if the inconsistency experienced by operations in the schedule (compared to if operations were executed serially) is within the specified inconsistency allowed for the operation. A weakness with [Won92] and [Pu91a] is that they seem to be applicable only in the narrow domain consisting of applications involving numerical quantities [Meh93].

4.4.3 Relaxing atomicity

In order to preserve the autonomy of local sites, various commit protocols based upon relaxing the atomicity requirement have been proposed.

Synopsis 4.30 $\Longrightarrow$ Gray's and Garcia-Molina's compensating transactions

Another option is to relax the atomicity properties of transactions. These schemes rely on the notion of *compensating transactions* [Gra81]. Compensating transactions reverse the effect of a committed transaction. A compensating transaction restores database consistency by undoing the effects of a committed transaction and results in a weaker notion of atomicity - *semantic atomicity* [Gar83]. Compensating transactions are also used in [Gar87] to amend partial executions of sagas [Meh93]. A comprehensive treatment of compensation can be found in [Kor90].

Synopsis 4.31 $\Longrightarrow$ Levy *et al*'s isolation of recoveries scheme

Ensuring atomicity of global transactions in a distributed environment implies loss of local autonomy at local sites, long duration delays and blocking. In [Lev91a], the authors deal with these problems by proposing an *optimistic 2PC protocol* in which locks are released as soon as a site votes to commit a transaction. If, finally, the transaction is to be aborted, then its effects are undone semantically by a compensating transaction. In [Lev91a, Lev91b], correctness criteria are proposed that prevent unacceptable executions when atomicity is given up for semantic atomicity. The authors note that if there are transactions that do not satisfy the all-or-nothing atomicity property in the system, then other transactions may see the partially committed effects of the transaction which may be unacceptable. To prevent this, the authors introduce the correctness criteria of *isolation of recoveries* (IR). A schedule is in IR if no transaction sees both the compensated-for effects, as well as the committed effects of other transactions. Thus the IR execution prevents transactions from seeing certain inconsistent states of a database. Further, protocols to ensure that the resulting schedules are IR are developed under the assumption that each site follows a strict 2PL protocol for concurrency control. [Meh93]

4.4.4 Other approaches

Synopsis 4.32 $\Longrightarrow$ Korth's and Herlihy's exploitation of operation semantics

Another approach to enhancing concurrency is to retain the serializability requirement, but to exploit the semantics of operations richer than primitive read and write operations when defining conflicts. In [Kor83], the author generalized read and write locks to a set of lock types that offer different degrees of exclusion based on the semantics of operations. In [Her90], the author defines a conflict between two operations not on the basis of whether they commute[5], but based on whether the exclusion of one invalidates the other. The paper proposes new optimistic concurrency control techniques for objects in distributed systems, proves their correctness and optimality properties, and characterizes the conditions under which each is likely to be useful.

Synopsis 4.33 $\Longrightarrow$ Weihl's commuting operations

In [Wei88, Wei89], the author proposes a pessimistic scheme that allows concurrent operations to update the same entity as long as the updates commute. The conflict relation between operations is defined differently depending on the recovery method being adopted. This scheme permits the results returned by operation executions, as well as their names and arguments, to be used in determining the conflict relation. This allows a greater degree of concurrency control while still maintaining serializability [Ras93b].

Synopsis 4.34 $\Longrightarrow$ Badrinath & Ramamrithan's recoverability

In [Bad92], the authors identify a property known as recoverability which is used to decrease the delay involved in processing non-commuting operations while still avoiding cascading aborts. An invoked operation that is recoverable with respect to an uncommitted operation can commit even if the uncommitted operation aborts. Since performing recovery is complicated here, in [Ras93a], the authors extend the notion of strictness to schedules containing operations richer than just reads and writes. Also commutativity between operations and operation inverses is utilized in order to develop schemes that ensure schedules are strict [Ras93b].

Synopsis 4.35 $\Longrightarrow$ Shasha *et al*'s partitioning of transactions

In [Sha92], an algorithm has been proposed that partitions global transactions into subtransactions which can be interleaved arbitrarily and the resulting schedule will always be serializable. This approach differs from sagas because not all interleavings of sagas will produce serializable schedules. This approach is applicable in environments where the set of transactions that can run is known in advance [Ras93b].

[5] do not conflict

Researcher	Scheme Name	Synopsis
Barker & Özsu	Basic MDB model	4.22 — Serializability graphs
Pu	Superdatabases	4.15 — Violates local autonomy
Breitbart *et al*	Replicated data model	4.2 — Site graph 4.8 — Rigorous schedules 4.10 — Partitioning
	Server model	4.5 — Optimistic ticket method
Elmagarmid *et al*	Stub approach	4.3 — Serialization events
Chen *et al*	Distributed MDMS	4.24 — Two phase locking & Linear ordering of resources
Kang & Keefe	Decentralized GTMs	4.11 — Distributed strict timestamp ordering
Garcia-Molina & Salem	Sagas	4.25 — No scheme needed
Yoo & Kim	Client server Approach	Not addressed

Table 4.1: Concurrency control — core group transaction management schemes

Discussion

In much of the work on recovery by compensation [Meh93, Gra81, Gar87, Lev91a, Lev91b, Elm90b], it is assumed that a compensating transaction can be associated with the original transaction to semantically undo the effects of the transaction. The issue of conditions under which compensating transactions exist and the related issue of designing compensating transactions is not addressed. In contrast, in the work by [Kor90] the authors identify sufficient conditions under which a compensating transaction is possible.

4.4.5 Summary

Table 4.1 gives a summary of the concurrency control mechanisms which are used by the transaction management schemes in our core group.

4.5 Global Deadlock Detection

Very little research has been done into the global deadlock problem in multidatabases. There are as yet few schemes that preserve local autonomy and global serializability while still maintaining an acceptably high level of concurrency. Previous mechanisms for deadlock resolution can be summarized as in Table 4.2 [Nam93, Tun92].

The mechanisms can be categorized according to three approaches.

- Firstly, the *no-wait* approach attempts to break the waiting conditions which could

Approach	Researcher	Reference	Mechanism	Global Serializability	Local site Autonomy	Degree of Concurrency
No-wait	Gligor *et al*	[Gli86]	Do almost nothing	No	No	High
	Gligor *et al*	[Gli86]	Homogenous LTM's	Yes	No	Low
Deadlock prevention	Gligor *et al*	[Gli86]	Off-line updates	Yes	No	Low
	Kim *et al*	[Kim92]	Wait-die	Yes	Yes	Low
	Vidyasankar	[Vid91]	Rooted tree access	Yes	Yes	Low
	Barker	[Bar90]	Total ordering	Yes	No	Low
Deadlock detection	Sugihara	[Sug87]	Distributed cycle-detection	Yes	No	High
	Breitbart *et al*	[Bre90a]	Potential conflict graph	Yes	No	Low

Table 4.2: Global deadlock resolution mechanisms in MDMSs

cause deadlock occurrences, so that no more deadlock resolution is necessary. Two mechanism exist within this approach:

- the *do almost nothing* mechanism in [Gli86] does nothing, but forces data to be released immediately after the execution of each operation of global transactions, and
- the *off-line updates* mechanism in [Gli86] allows only off line sequential updates.

- Secondly, the *deadlock prevention* approach essentially orders the way in which transactions claim locks, so that cyclic waiting never occurs between global transactions. Three mechanisms take this approach:
  - the *homogenous local transaction managers* mechanism in [Gli86] generates only the equivalent serializable execution schedules without the cyclic waiting in every LDB,
  - the *wait-die* mechanism in [Kim92] only allows an older transaction to wait for a younger transaction when conflict occurs between them and not visa-versa,
  - the *deadlock-free concurrency control scheme* in [Vid91] allows data access only in a rooted tree fashion,

  - Barker's [Bar90] scheme which orders global transactions in a total order.

- Thirdly, the *deadlock detection* approach checks for cycles in the wait-for graph of transactions so that deadlocks can be detected explicitly.

  - One mechanism takes this approach [Sug87], the *distributed cycle-detection* maintains a local serialization graph in each local database system so that distributed cycle detection is possible.
  - Breitbart [Bre90a] uses a potential conflict graph (PCG) which is constructed using all the global transactions in the system. The GTM uses a timeout scheme. If no response is obtained from a transaction within a certain time, then the PCG is checked for cycles. If a cycle is found, the youngest transaction in the cycle is aborted.

All eight methods outlined have significant drawbacks [Nam93]. The *do-nothing* mechanism could cause unserializable execution histories if the data is released immediately after each operation.

The *homogenous local data manager* and *distributed cycle-detection* mechanisms fail to preserve local autonomy since in the former the local data manager has to be modified and in the case of the latter the local deadlock resolution mechanisms will have to be altered so that the global serialization graphs can be managed.

The *off-line updates, deadlock-free concurrency control,* and *wait-die* mechanisms the degree of local concurrency is seriously hampered. In the case of *off-line updates*, the data may only be updated off-line and serially; in the case of *deadlock-free concurrency control scheme* the data is only accessed in rooted-tree fashion and in the case of *wait-die*, unnecessary restarts could be caused due to non-real deadlocks. Breitbart's method [Bre90a] does not allow global update of locally updateable data items, does not process operations of the same transaction concurrently and the cycle detection algorithm wastes a lot of time because it will be activated by all the blocked transactions in a cycle at the same time. This scheme also runs the risk of declaring deadlock without any confirmation and thus transactions could be unnecessarily aborted [Tun92].

Nam & Moon [Nam93] have come up with a method for performing global deadlock detection without violating local autonomy or global serializability. Nam & Moon [Nam93] propose that the *global deadlock detector* (GDD) itself must construct a local wait-for graph at each local participating database site. It must then combine all these graphs and construct a global wait for graph to see if any cycles exist. They prove that if the GDD is implemented at each site on top of the local LDBS, local autonomy and global serialization can be maintained while also allowing a high degree of concurrency.

The scheme in [Bat92] also maintains local autonomy and global serialization but could cause an unacceptable number of aborts although it is fully decentralized and therefore more fault tolerant than the scheme proposed by [Nam93] which requires active participation from

the central site in order to succeed.

Tung [Tun92] proposes a scheme to control global deadlock in a multidatabase too. In their scheme, a Transaction-Blocked-at-Site-Graph(TBSG) is defined. It is an undirected graph where T is the set of currently blocked global transactions and S is the set of sites currently being accessed by these transactions. An edge in the graph is defined between T_i and S_j if transaction T_i is currently blocked and accesses S_j. Tung [Tun92] proves that if a global deadlock exists, then there must be a cycle in the TBSG; provided there are no local deadlocks at any local site. They also prove that global deadlock cannot exist if the TBSG is acyclic. The scheme strives to reduce the possibility of false global deadlocks, and attempts to minimize the recovery costs by effective choice of a victim transaction by the use of a heuristic algorithm in the case of a deadlock. This will reduce the possibility of livelock too. The recovery scheme fully preserves local autonomy. It utilizes the original local database recovery procedures and the servers which are located at the local databases to provide a simulated 2PC protocol to ensure the global consistency in the event of transaction failures, site failures and MDMS failures.

After studying the previous research into deadlock control, we can come to the following conclusions [Nam93]:

1. If a GTM is used by the MDMS, the local DBMSs can increase their performance by preventing local deadlocks due to direct conflicts between uncommitted global transactions. The GTM complicates global transaction management procedures:
   - There are additional costs in maintaining the GTM in the MDMS.
   - The multidatabase deadlock detection procedure is fairly complicated
   - The recovery procedure also becomes more complicated.
2. Without knowledge of local DBMS schedules, deadlock can be prevented if all global transactions are executed in serial order at the MDMS level. This decreases concurrency drastically.
3. Higher concurrency can be obtained by using an optimistic concurrency control method and then aborting problematic transactions. This causes a high level of aborts.
4. Without a synchronization point to synchronize executions of global transactions, it is very difficult to maintain global consistency.
5. It would be a great advantage if all local sites could agree on some or other predefined protocol (e.g. 2PC) to commit global transactions.

The global deadlock characteristics of the concurrency control mechanisms used by the transaction management schemes in our core group can be summarized in Table 4.3.

Researcher	Scheme Name	Deadlock control
Barker & Özsu	Basic MDB model	Prevents deadlock
Pu	Superdatabases	Not addressed
Breitbart *et al*	Replication model	Ensure that the global site graph is acyclic and assume that local schedules are deadlock free
	Server model	Uses a global *wait-for* graph and checks for cycles
Elmagarmid *et al*	Stub approach	Deadlock free
Chen *et al*	Distributed MDMS	Deadlock free
Kang & Keefe	Decentralized GTMs	Deadlock free
Garcia-Molina & Salem	Sagas	Not addressed
Yoo & Kim	Client server Approach	Detect and resolve

Table 4.3: Global deadlock in the core group transaction management schemes

4.6 Summary

In this chapter we have introduced the special problems encountered in transaction management, global concurrency control and global deadlock management in multidatabase systems.

Transaction management in general and the functions of the global transaction manager were discussed. The formal transaction model was extended to include multidatabase concepts. A brief synopsis was given of work by researchers in the core group and an outline given of the transaction management scheme and global concurrency control characteristics of each scheme. The general problems inherent in integrating various concurrency control methods are also addressed. An overview was also given of the research done into global concurrency control by various researchers in the field. The global deadlock detection aspect of multidatabase transaction management was briefly discussed and the methods used by the core group for global concurrency control and global deadlock detection are summarized.

The next chapter will discuss the reliability aspect of transaction management in multidatabases.

Chapter 5

Reliability

Barker & Özsu [Bar91] define reliability as comprising two parts: transaction atomicity and crash recovery.

- **Transaction atomicity** means that the effects of committed transactions are reflected on the database, but the effects of uncommitted or aborted transactions do not appear.

  Much of the research done into transaction management in multidatabases assume that no failure occurs during transaction processing [Alo87, Bre91a, Bre88, Elm90a, Geo91b]. In a failure free environment, serializability of global transactions can be guaranteed easily by the strict two-phase locking protocol of the underlying local database systems. If, however, this is not the case, then a failure might cause the unilateral abort of a subtransaction at a local database system. This happens because local database systems cannot be expected to participate in a global two-phase commit protocol [Ber87, Özs91]. Hence a simulated global commit protocol is needed in a multidatabase environment [Yoo95].

  A global commit and recovery protocol must deal with the following events due to the autonomy of the local database systems [Yoo95]:

  - *Unilateral abort of subtransactions due to site or LDBMS failures*: LDBMSs cannot distinguish local transactions from global subtransactions so when it recovers after a site failure, its local recovery procedure rolls back all uncommitted subtransactions as well as uncommitted local transactions. It makes no difference that the global transaction that the subtransaction belongs to may have committed.
  - *Unilateral abort of subtransactions due to commit operation failures*: A subtransaction may fail at the commit operation in an LDBMS, even after its database access operations are successfully executed in the LDBMS. This can cause a globally inconsistent state to occur.

- *Exposition of the incomplete results*: From an LDBMS viewpoint, the recovery action of the MDMS is also a transaction that has no connection with the failed subtransaction. Thus, exposition of incomplete results to other transactions may occur after a unilateral abort of a subtransaction occurs, but before recovery action is started and successfully done.

- **Crash recovery** requires that in the event of a system failure, the database is recovered to a consistent state so that transactions terminate according to the transaction atomicity condition [Bar90].

This chapter will deal with ensuring transaction atomicity in multidatabase systems. Chapter 6 deals with the recovery question.

5.1 Transaction Atomicity

In a multidatabases system, various local database systems are integrated and each may support a different commit protocol.

The problem of how to satisfy the requirements of 2PC in a multidatabase is often not addressed in the literature. The basic requirement which must be satisfied in order to develop a variation of 2PC for a multidatabase environment is the availability of a *visible prepare to commit state* for all subtransactions of global transactions [Geo90]. A subtransaction enters its prepare to commit state when it completes the execution of its operations and leaves this state when it is committed or aborted. Only when a subtransaction is committed are its updates installed in the database. The prepared state is *visible* if the MDMS can decide whether the subtransaction should commit or abort [Geo90].

Some transaction managers have an open commit protocol; which means that local transaction managers can participate in the commit decision and that their commit protocols are public. However, many commercial transaction managers have a closed commit protocol, in that transaction managers cannot participate in a decision to commit. Closed transaction managers are systems which have private protocols and therefore cannot cooperate with other transaction managers.

Several popular transaction processing systems are closed — among them IBM's IMS and Tandem's TMF. On the other hand, many commercial DBMSs provide primitives to support a visible prepare to commit state for each subtransaction. For instance, the Remote Data Access (RDA) [Ber90] standard and many DBMSs designed using the client server architecture (e.g. SYBASE) provide primitives that allow applications to inquire about the status of database operations they submit. The MDMS can then determine whether database operations of subtransactions have been completed and then also when each subtransaction enters its commit state.

If transaction managers of the local database systems are open and can participate in some form of two-phase commit protocol, then it is possible to integrate the various

protocols into some sort of two-phase commit protocol.

On the other hand, it is virtually impossible to implement general ACID global transactions involving closed transaction processing monitors. The key problem is atomicity: the closed transaction manager can unilaterally abort any subtransaction, even though the others decide to commit [Gra93].

In conclusion, we cannot assume that a *two-phase commit protocol* is available because the component databases may not have that facility and anyway it would violate local autonomy so that we must assume that the databases which constitute a MDB:

- do not communicate with each other,
- do not synchronize, and
- must maintain their own autonomy (as far as possible).

The only way we can ensure atomicity is to be able to guarantee that the operations of each subtransaction can be submitted to the underlying DBMS in a *separable* manner [Sop91b]. We now have three options to ensure transaction atomicity in a multidatabase:

- One means of emulating an atomic execution is to attempt to commit a global transaction by *resubmitting the corresponding subtransaction* at each site where it erroneously aborts. This approach involves issues of serializability [Sop91b].

  The approach of re-submitting the subtransaction requires placing restrictions on the access patterns of the global and local transactions to preserve local autonomy and provide the ACID properties. One approach would be that at each site the data could be partitioned into global and local sets (see Synopsis 4.10). Unrestricted access would then be available to global and local transactions only with respect to their respective sets. Access across the sets is restricted so that a resubmitted subtransaction faces no contention for data. This is a severe restriction.

  The restriction can be relaxed to a certain extent if the MDMS is allowed to abort subtransactions at a local site. If a subtransaction is to be aborted, then active subtransactions which need to be run serially after that transaction must also be aborted.

  Yet another option is to violate control autonomy when resubmissions are done. Any local transaction which can interfere with the commitment of a subtransaction is blocked until the subtransaction is either committed or aborted. This approach requires the transactions to declare the data they access prior to their execution - a major violation of local autonomy.

- An alternative is to try to approximate the effects of aborting a global transaction by *submitting a compensating subtransaction* at each site that committed the subtransaction where it should have aborted. The problems here are that the semantics of the

transactions need to be taken into account, and it is often impossible to design such a compensating transaction.

- Another case is where the local DBMS allows the *submission of a commit or abort operation separately from the body of the transaction.* In this case it is simple to maintain transaction atomicity. If the local DBMS then accepts transactions in which commit and abort operations are separated from the body of the transaction, and violates control autonomy, we can use typically distributed DBMS techniques to guarantee atomicity. The well known two-phase commit can be extended to the multidatabase environment (See Appendix C). Multidatabases do not enjoy the luxury of inter-database system communication and synchronization which makes two-phase commit more difficult [Bar90].

Example 5.1 *:* Problems with submitting commit and abort operations separately

Suppose there are two global transactions GT_1 and GT_2 in a multidatabase system executing two-phase commit simultaneously. If the global transactions have participants in common, the phase two commit messages might arrive in different orders at the common sites. Hence at one site GT_1 may commit before GT_2 and at another site GT_2 will commit before GT_1. This has an impact on global serializability [Ber93]. ◇

In DBMSs that do not support a prepare to commit state, the following alternative approaches can be used to satisfy this requirement [Geo90]:

- Modify the local DBMSs to provide the necessary primitives [Pu88]. This seriously violates local autonomy and is not acceptable.

- Use a mechanism that forces a handshake after each transaction operation [Geo91b, Bre90a]. With this approach the MDMS will submit the operations of a global transactions one at a time and wait for completion thereof before submitting the following operation. This approach does not violate local database autonomy but forces a total order onto the operations of a subtransaction.

- Design subtransactions in such a way as to simulate a prepared-to-commit state [Mut91]. This requires the use of inter-process communication primitives which once again may not be available.

- Emulate a two-phase commit process by using an *agent* process at each site. During the commitment of a global transaction, the MDMS acts as coordinator and the MDMS agents at the local databases act as participants [Tan93].

Definition 5.1 — *Correctness of atomic commitment protocols*

We define an atomic commitment protocol (ACP) to be correct if for each global transaction (GT_i) submitted to the GTM, the GTM:

1. uniformly commits or aborts all subtransactions of GT_i in a finite amount of failure-free time,
2. preserves database consistency. Since we assume that each transaction (when executed in isolation and with no failures) preserves database consistency, it must do so when using the ACP,
3. commits all subtransactions if no other global or local transactions are currently executing and there are no failures.

[Mul92] □

Conditions 1 and 2 are standard. The third condition is made to exclude protocols of the form: "abort all transactions" or "abort all transactions except read-only transactions" from consideration. That means that the ACP must not limit the class of transactions it accepts. Any transaction that runs in isolation and without failures should commit.

Bearing this in mind, Mullen *et al* [Mul92] have proved that it is impossible to implement an atomic commitment protocol without violating local autonomy in multidatabase systems. This is true even in the absence of system failures. Mullen *et al* [Mul92] have also shown that even if one were to assume that all local DBMSs use strict two-phase locking as their concurrency control method, atomic commitment is impossible if even a single system failure occurs.

We now have to devise a solution to the global commitment problem in multidatabase systems by using various different strategies. In the following section we will take a look at how the core group handles this problem.

5.2 Global Commit Protocols in the Core Group

The transaction management schemes in our core group use different global commit protocols. In this section a brief outline is given of the method used in each scheme.

5.2.1 Barker & Özsu's transaction atomicity scheme

Barker & Özsu [Bar91] introduce a model which maintains that global transactions and subtransactions have certain states and which gives a technique whereby these state transitions can be managed and whereby they emulate a two-phase commit without affecting local autonomy.

Global transaction states : When a global transaction is submitted to the MDMS, it is in an *initial* state. Once all GSTs of global transaction GT_i have been submitted to their respective DBMSs, GT_i is moved to the WAIT state. If all GST's become ready GT_i is moved to the *commit* state. If any one of the GSTs do not become ready GT_i moves to the *abort* state.

Global subtransaction states : Once a GST is submitted to a DBMS it is in the *initial* state. The GST remains in this state until it decides to *abort* or it is ready to commit, in which case it moves to the *ready* state. Once the GST is in the *ready* state, it waits for the final commit decision from the MDMS. If GT_i commits, a message is sent to the local level and the GST can move to the *commit* state. If GT_i is aborted, the GST moves to the *abort* state.

The technique : The difficulty lies in maintaining the state transitions at the local level. The MDMS needs a *ready* state to coordinate the termination of all the global transaction's subtransactions. If the component DBMSs implement a one-phase commit, however, they do not provide a *ready* state. Because the local DBMSs are autonomous, no modification of their protocols is possible, so some technique to emulate a two-phase commit is now proposed:

GST_i^j does various operations and then it decides whether to abort or commit. If it decides to commit, we modify the commit operation by making GST_i^j send a signal to the MDMS and wait for a decision from the MDMS and then either commit or abort depending on that reply. It will block till a response is received but this does not affect local transactions at all. Barker & Özsu [Bar91] have proved that the local DBMSs need to guarantee a locally strict level of service so that while a GST is blocked and waiting for a response from the MDMS, another transaction will not be allowed to access data items altered by the GST. This will be done by the local DBMS and the MDMS does not have to intervene to ensure this.

5.2.2 Pu's hierarchy of superdatabases

Pu [Pu88] states that any agreement (commit) protocol will do for the superdatabase system. We could use two-phase, three-phase commit or byzantine agreement.

In the superdatabase system, at each level the parent transaction serves as the coordinator. During phase one, the root sends the message "prepare to commit" to its children. The message is propagated down the tree, until a leaf subtransaction is reached, when it responds with its vote. At each level, the parent collects the votes; if all of its own children voted "yes", then it sends "yes" to the grandparent. If every subtransaction voted "yes", the root decides to commit and sends the committed message, propagated down the tree. Between the sending of the vote and the decision by the root, each child subtransaction remains in the prepared state, ready to undo the transaction if aborted or redo if the child

crashed and the root decided to commit.

5.2.3 Breitbart *et al*'s work

Breitbart *et al* [Bre90a] use the two-phase commit protocol. When the MDMS encounters the commit operation of global transaction GT_i, it sends a *prepare to commit* message to each server involved in the execution of GT_i. Each server receiving the message determines if it can commit the subtransaction belonging to transaction GT_i. If it can commit, it forces all the log records for the subtransaction to stable storage, including a *ready* record. It then notifies the MDMS whether it is ready to commit or whether GT_i must be aborted. The MDMS collects all responses and if all have voted *ready*, the MDMS will commit GT_i. If at least one server voted to abort or fails to respond, the MDMS aborts GT_i and notifies all servers involved with GT_i.

Each server, upon receiving the decision of the GTM, informs the local DBMS as to whether to abort or commit the global subtransaction at that site. It is thus obvious that this model expects the local databases to support the prepare-to-commit state. This is an unreasonable assumption as many commercial database systems do not provide this capability.

5.2.4 Elmagarmid *et al*'s work

Elmagarmid *et al* [Elm90a] do not make provision for failures in their model and therefore have no need of an atomic commit protocol.

5.2.5 Chen *et al*'s distributed MDMS

Chen *et al* [Che93] use a semantic based commit method. Chen *et al* decompose a subtransaction into two steps, an *execution step* and an optional *confirm step* and an optional *undo step*. These steps are determined from the semantics of the subtransaction. The following guidelines are given for defining these steps:

- For a *compensatable* subtransaction, the execution step includes all the operations of the subtransaction, and the undo step contains all the compensating operations for the subtransaction. No confirm step needs to be defined.
- For a *read-only* subtransaction, only an execution step needs to be specified. In this case the execution step consists of all operations in the subtransaction.
- For a *noncompensatable* subtransaction GST_j^i of global transaction GT_j the situation is more complex. Depending on the nature of GST_j^i, two options are available:
  - If GST_j^i must be run on an autonomous local database system, the execution step includes all the operations of GST_j^i from the beginning up to the prepare-to-commit state of GST_j^i. The confirm step includes all operations from the

prepare-to-commit state up to the commitment of GST_j^i. The undo step contains all operations from the prepare-to-commit state up to the aborting of GST_j^i. The execution step completes first. During commitment of GT_j, if GST_j^i must be committed, the confirm step is executed; otherwise the undo step is executed. In this way GST_j^i can be executed as two separate local transactions.

- If GT_j and GST_j^i are capable of communicating, allowing GT_j to control the execution of GST_j^i, then the execution step is defined as GST_j^i itself. At the prepare-to-commit state, GST_j^i will wait for a signal from GT_j after reporting its status to GT_j. The confirm step in this case will simply be used to commit GST_j^i, and the undo step is used to abort GST_j^i. In this way GST_j^i is executed as a single local transaction. This is very similar to the two-phase-commit protocol. This approach would be used for an underlying DBMS which provides a visible prepare-to-commit state.

This commit method has the advantage that its semantic structure allows application programmers to customize their own necessary commit decisions. They can be guided by the semantics of the global subtransaction and the commit methods of the underlying systems, using a uniform syntax for differing commit protocols [Che93]. This commit protocol could be used in both nested and multi-level transaction environments [Gra93]. This method is more flexible than those proposed in [Kor90, Lev91a, Lev91b] which are designed for multi-level transaction environments only.

5.2.6 Kang & Keefe's decentralized GTMs

Kang & Keefe [Kan93] assume that an atomic commit protocol such as 2PC is employed. For each global transaction there is one coordinator and several participants which all contribute towards reaching a consensus on the commit decision for the global transaction. When global transaction GT_i finishes its operations at the coordinator site, the coordinator sends each participant a *prepare* message. If the participant can commit, it force writes all the local redo records of GT_i and a *prepared record* is sent to the GTM's log at the local site. The *prepared record* indicates that GT_i is in the prepared state. It then sends a *ready* message to the coordinator.

If all participants respond with *ready*, the coordinator issues a *commit* message to all participants. If any participant responds with a *veto* message, or fails to respond within the timeout time, the coordinator issues an *abort* message to all participants. The participant leaves the ready state once it has either committed or aborted [Kan93].

If a prepared transaction is aborted by the LTM, it is the responsibility of the GTM to determine the fate of the subtransaction and resubmit its redo transaction when necessary. Because Kang & Keefe assume that the LTM does not participate directly in the protocol, that means that the LTM can abort a prepared subtransaction either by a unilateral decision (e.g. timeout) or due to failure at the local site. This would mean that all the resources

currently assigned to the subtransaction would be released. If the global decision was to commit, the GTM must redo the updates done by the transaction by submitting a redo transaction. The problems this causes have been discussed previously but Kang & Keefe deal with it by partitioning data into globally updateable and locally updateable groups.

Kang & Keefe have designed a commit protocol which makes provision for query subtransactions, that is, read-only subtransactions. These transactions need not be subjected to the strict controls applied to update transactions and Kang & Keefe's model addresses this. Kang & Keefe's partitioning of data objects approach differs from Breitbart *et al*'s [Bre90a] because it allows read-only subtransactions to *read* locally updateable data objects.

They have also formulated a *cautious presumed-commit protocol.* In line with this protocol, read-only subtransactions may commit without waiting for the commit message from the coordinator. An abort message from the coordinator is ignored because no changes were made to the database by the read-only subtransaction. If such a transaction is locally aborted, it sends a veto message to the coordinator.

Kang & Keefe prove the following theorem [Kan93]:

Theorem 5.1 *: Global reliability of a GTM*

A GTM is globally reliable if the following conditions are satisfied:

1. *A global subtransaction which unilaterally aborts does not read locally updateable objects; and*
2. *Every global subtransaction, including redo transactions, contain the take-a-ticket construct and no local transaction can write the ticket.*
3. *The GTM only outputs conflict serializable and strict executions of global transactions; and*
4. *The local transaction manager only outputs cascadeless one-copy serializable histories with a reasonable version function.*

△

5.2.7 Garcia-Molina & Salem's sagas

The saga scheme [Gar87] doesn't use a global commit protocol because the transaction model does not need one. The application program issues various commands to the system: begin-saga, a series of begin-transaction, end-transaction commands and finally an end-saga command. There is also an abort-saga command which is sent out to abort all transactions of the saga. The transactions of a saga are aborted by means of compensation.

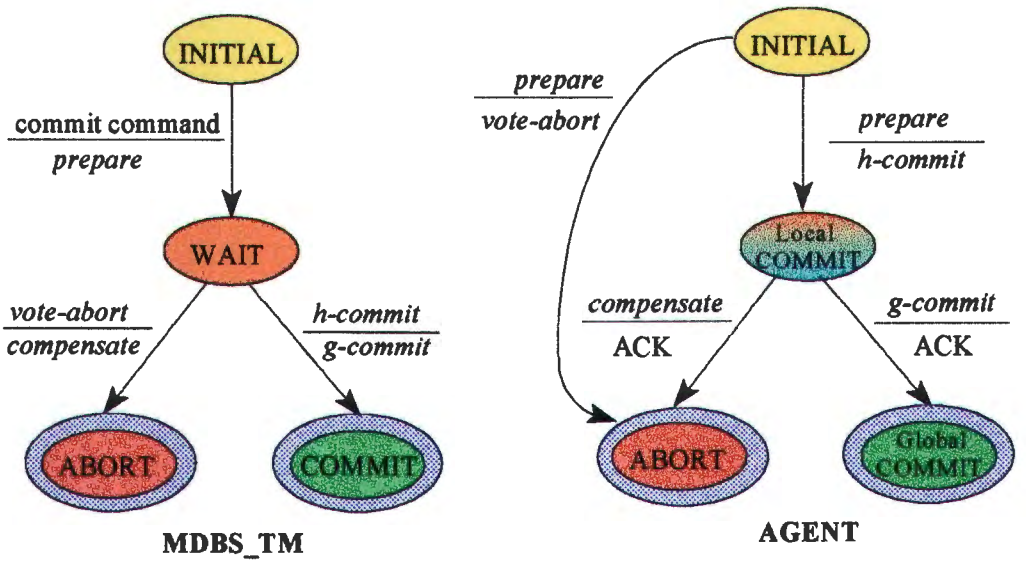


Figure 5.1: State transition in the R2PC protocol

[Yoo95, p58]

5.2.8 Yoo & Kim's client server approach

Each stub in Yoo *et al*'s scheme [Yoo95] controls the submission of operations issued by global subtransactions and local transactions using stub-level locks and a stub-level locking table. Only update operations have to obtain stub-level locks. Stub-level locks are granted on a first in first out basis and the locks held by a global subtransaction are released when a global transaction that includes the global subtransaction commits or aborts or the subtransaction is aborted.

Yoo *et al* propose a resilient global atomic commitment technique called a *reliable two-phase commit* (R2PC) protocol. The R2PC protocol consists of commit, termination and recovery protocols. R2PC guarantees fault-tolerant global atomicity in multidatabase systems where the local DBMSs have no prepare-to-commit state. This approach does not simulate the prepared state. State transitions in the R2PC protocol are shown in Figure 5.1.

- **MDBS_TM's Commit Procedure**

  1. The MDBS_TM writes a *prepare-record* in its log and sends a *prepare* messages to all agents that participate in the execution of that global transaction and waits

for participant's votes.

2. If every vote from participating agents is *heuristic commit*, MDBS_TM writes a *commit-record* in the log, returns a success message to the user and sends a *global commit* message to participating agents and waits for *ack* messages from them.

   If even one message is *vote-abort*, MDBS_TM writes a record to the log, returns an error message to the user, sends a compensate message to the agents that voted to commit and waits for acknowledgement of the messages.

3. When *ack* messages are received from all the participating agents, MDBS_TM finally writes an end-of-transaction message to the log and forgets about the transaction.

- **Agents Commit Procedure**

1. After receipt of a prepare message from the MDBS_TM, each agent writes log-records to build a compensating transaction into the log, writes a *ready-record* to the log, saves stub-level locking information to the log and submits a commit command to its LDBMS.
2. When the subtransaction commits successfully in the LDBMS, the agent replies with a *heuristic commit* message to the MDBS_TM and waits for a final decision. Otherwise, the agent releases all locks held and replies with a *vote-abort* message.
3. If the final decision by the MDBS_TM is to commit, the agent releases all locks and writes a *commit-record* message to the log. If the final decision is to abort, the compensating transaction will be carried out.

These protocols work hand in hand with timeout protocols (full details of timeout protocols can be found in [Yoo95]). This occurs at a destination when a site cannot receive an expected message from a source site within a specified timeout period.

5.2.9 Other relevant research

5.2.9.1 Georgakopoulos's simulated prepared to commit state

Georgakopoulos [Geo91a] proposes using a *simulated prepared to commit* state. Georgakopoulos submits subtransactions an operation at a time to the local DBMS so the GTM knows precisely when all operations have been completed. The GTM then knows whether the subtransaction wants to commit or abort. The basic difference between the traditional prepared to commit state and the simulated prepare to commit state is that a transaction in the simulated state has no assurance from the DBMS that it will not be aborted unilaterally. However, Georgakopoulos claims that DBMSs do not unilaterally abort transactions that have entered their prepared to commit states because by then they have completed all their operations and acquired all their locks.

Georgakopoulos states that if all the subtransactions of a global transaction have reached the simulated prepare to commit state, the GTM can then submit the commit operations to the local DBMSs and the global subtransactions all commit at the local sites.

However, he fails to address the problem of transaction timeouts which could well cause a local DBMS to abort a prepared transaction, in which case global consistency would be violated.

5.2.9.2 Perrizo *et al*'s atomic commitment

In the HYDRO (HeterogeneouslY Distributed Request Ordering) system, the authors use a variation of the standard 2PC protocol [Per91]. In the HYDRO system, the transactions have to declare all their data needs in advance and then the global transaction is parsed into subtransactions and these subtransactions are submitted to local database systems. The local DBMS is expected to notify the MDMS when it commits or aborts a global subtransaction. Until this notification has been received, the MDMS keeps the state of the global transaction as *prepared.* If the local DBMSs all commit the global subtransactions, the state of the global transaction would be changed to *commit.* If the GTM decides to abort a global transaction, a compensating transaction will be sent to all sites where subtransactions of that transactions have already committed.

In order to maintain global database consistency in the face of compensating transactions, HYDRO blocks all other transactions until a global transaction which has an active subtransaction at a certain site has committed. This is a severe restriction which reduces concurrency to almost nil.

5.3 Analysis

There are various approaches to the global atomic commitment problem in multidatabase systems.

From the preceding discussions we can see that it is basically impossible to achieve global atomicity in the face of failures without somehow violating local autonomy. Because of this, most works in the literature allow certain tradeoffs, i.e. sacrifice one or more of the desired features of an ideal MDMS [Yoo95]. The commonly used assumptions are, for example, no subtransaction failure, no inter-subtransaction data dependency, and partitioning of the database into two or more sets so that global and local transactions access these sets exclusively.

Barker & Özsu [Bar91] use the notion of m-serializability as the correctness criterion instead of global serializability and assume that local database systems produce only serializable and strict schedules. The authors propose a simulated two-phase commit and do not deal with a unilateral subtransaction abort.

Kim *et al* [Kim93] also assume that all the LDBMSs use strict 2PL. Their approach

is similar to Barker & Özsu's approach except for log management. Whereas in Barker & Özsu's scheme the LDBMS will always block after failure even if there is no subtransaction to be resubmitted, Kim *et al*'s approach solves that problem but in Kim *et al*'s protocol, the LDBMS can block if a MDMS fails during the exclusive access period. Kim *et al* also assume that the LDBMS restarts after failure in exclusive mode, which prevents local transactions from being restored until after the MDMS is finished — which violates local autonomy. Kim *et al* also do not deal with unilateral subtransaction abort due to commit operation failure.

Pu [Pu88] uses a two-phase-commit in which all "leaf" nodes are expected to participate — which violates local autonomy. In [Bre90a], Breitbart *et al* partition the data into two separate sets and also assume that the local DBMSs support a prepare-to-commit state. In Breitbart *et al*'s later work [Bre92b], it is assumed that the LDBMSs use strict 2PL. A server at each site will facilitate the 2PC required for global atomicity. This is considered reasonable by some authors but the fact remains that it may not be available in all local database systems of a multidatabase and the failure of only one DBMS to provide such a state would make the whole commit protocol collapse. The partitioning required in this case would not always be practical.

Elmagarmid and Du [Elm90a] do not deal with the failure question. Chen *et al* [Che93] propose a semantics based commit method which is very attractive except that they assume that a transaction can be broken up into portions, an execution step, an optional undo step and an optional confirm step. The application programmer would be expected to do this which could be quite cumbersome and failure prone.

Kang & Keefe [Kan93] also approach the problem by partitioning data into two sets and allow read-only transactions more leeway than is done in [Bre90a]. Kang & Keefe assume that cascadeless schedules are generated at local sites and use an agent at the different sites to implement a 2PC type of protocol which allows read-only transactions to commit without global permission.

Garcia-Molina *et al* [Gar87] do not make use of a commit method. Yoo *et al* [Yoo95] assume that local database systems use strict 2PL (also assumed in the scheme proposed in [Bre92b]) and that local database systems are disjoint. They also assume that only one subtransaction per global transaction is submitted per site. The latter two assumptions are in line with the transaction model outlined in Chapter 4 anyway. One could also argue that many commercial DBMSs use 2PL as a rule but certainly not all of them do.

Finally, Perrizo *et al* [Per91] present a scheme called HYDRO that is similar to Yoo *et al*'s scheme except for logging differences. The HYDRO multidatabase has been developed at North Dakota State University. In HYDRO, all transactions are expected to declare their data needs in advance, which is not practical. They also assume that no two subtransactions can execute concurrently in a local database even if they access different data items. Their approach may result in serial execution of global transactions with no global concurrency which is a waste of resources [Yoo95].

Georgakopoulos [Geo91a] assumes that local database systems produce serializable and strict schedules, and that the MDMS has exclusive access to the database in the case of failure — which of course violates local autonomy. They also simulate 2PC by means of an agent process at each site. This approach assumes that no unilateral subtransaction abort can take place which is an unreasonable assumption as such an event is not only possible but probable.

In [Sop91b], the authors trade control autonomy for reliability. They simulate the prepared state by means of an agent/server type process. In this protocol, an occurrence of subtransaction failure makes the set of active transactions that potentially conflict with the subtransaction to be forcibly aborted. This becomes a severe problem with long transactions. [Yoo95]

Levy *et al* [Lev91a] propose an optimistic commit protocol with semantic atomicity as its correctness criterion. The authors use compensating transactions and the scheme has no prepared state. This protocol does not block or become delayed like protocols that use 2PC, but the persistence of a compensation assumption cannot be implemented. This is because the MDMS has no knowledge of which local transactions access data items before compensation can take place. [Yoo95]

Mullen *et al* [Mul93] propose a reservation commitment scheme whereby global transactions have to pass through a reservation stage before being submitted to the local databases. The reservation stage will determine whether the subtransaction will be able to commit successfully at the local database. If all the subtransactions of a global transaction can commit, then the GTM submits them, otherwise the global transaction is aborted. The big drawback of this scheme is that local transactions have to be modified to fit in with this protocol which is an unacceptable violation of local autonomy.

Finally, we can conclude that in our core group Yoo *et al*'s and Chen *et al*'s global commitment schemes provide for maximal local autonomy. Yoo *et al* make more assumptions than Chen *et al* about the LDBMSs but even the ones that Yoo *et al* make are not unreasonable.

5.4 Summary

This chapter had a look at reliability in multidatabase systems. Reliability was defined as comprising transaction atomicity and crash recovery. Transaction atomicity was then elaborated upon while crash recovery was left for the following chapter. It was shown that transaction atomicity can be achieved only by the use of an efficient global commit protocol. The global commit protocols in the core group were examined and an analysis was given of the various approaches which are to be found in the literature.

Chapter 6

Recovery and Recoverability

Recovery can be defined as:

> Activities for ensuring that failures will neither infringe the atomic executions of global transactions nor corrupt persistent data [Tun92].

Recoverability can be defined as:

> The requirements that must be satisfied in order to guarantee correctness in case of failure [Geo90].

Multidatabase recovery is responsible for maintaining the atomicity and durability of global transactions in the presence of transaction, site and communication failures. While recovery in centralized databases has been well researched and many effective recovery protocols already exist, the autonomy and heterogeneity of local databases in a multidatabase make the recovery problem more complex. A serious problem in multidatabase recovery is that recovery in a multidatabase constitutes new transactions. A multidatabase recoverability condition has to be determined in order to assure that MDMS recovery can preserve global consistency [Geo90].

6.1 Failure in a Multidatabase

If a subtransaction of a global transaction fails it is automatically rolled back by the local database recovery manager, even if the MDMS considers the global transaction it belongs to as committed, and has allowed one or more subtransactions of the same global transaction to commit at other local databases. The MDMS cannot prevent site failures and unilateral subtransaction abortions by the local systems. The local autonomy of local databases does not allow any rollback of locally committed global subtransactions. To maintain global consistency the MDMS has to complete the failed subtransactions of global transactions which have a committed subtransaction [Geo90].

There are various causes of failure [Bel92, Bar91]: transaction failure, site failures, media failures, network failures, DBMS failures & system failures.

The actions required by the recovery manager following each type of failure is different and are discussed below. We are assuming that all systems are fail-stop; when failures or errors occur, they simply stop. We are excluding hardware and software bugs which cause the system to behave inconsistently.

6.1.1 Transaction failure

Transactions can fail either globally (at the MDMS level) or locally (at the local DBMS level). They fail in two ways: a value dependency test fails or a system requires the failure of the transaction for some other reason (e.g. resolving deadlocks). Transaction failures result in aborting the transaction and undoing its effects. The local failure of an individual transaction can happen for various reasons [Bar90]:

1. *Transaction-induced abort.* No other transactions are affected.

2. *Unforeseen transaction failure.* Arising from bugs in the application program. In this case, the system has to detect that the transaction has failed and inform the recovery manager to rollback the transaction. No other transactions are affected.

3. *System-induced abort.* Occurs when a recovery manager aborts a transaction because it conflicts with another transaction, or to break a deadlock. Again the recovery manager is explicitly told to rollback the transaction and other transactions are not affected apart from perhaps becoming blocked.

6.1.2 Site failures

This can occur as a result of the failure of a local CPU or as a power supply failure resulting in a system crash. All transactions on the machine are affected. We assume that both the DB itself on a persistent storage medium, and the log are undamaged. In a multidatabase environment, since sites operate independently, it is not only possible but probable that some sites can be operational and others failed. The main difficulty with partial failure is that sites should be able to determine the status of other sites. In a case like this it is possible for a site to become blocked and unable to proceed. For instance, say a site fails in mid-transaction. Other agents of the global transaction may be uncertain as to whether to proceed and commit or to rollback. In the context of a multidatabase it is imperative that the failures of certain sites not affect other sites [Bar90].

To recover from a site failure, the local manager must determine the state of the local system at the time of failure — more to the point — which transactions were active. The objective is to restore the DB to a consistent state by undoing or redoing transactions according to their status at the time of the failure by applying either before or after images from the log.

When the site is restored, control is passed to the recovery manager to execute recovery or restart procedures. During the recovery procedure, no new transactions are accepted until the DB has been repaired [Bar90].

6.1.3 Media failures

This is a failure which results from some portion of the stable database being corrupted due to something like a head crash. This type of failure will have to be handled by the local recovery manager according to the method decided on by the local data manager.

Recovery from local media failure is a local responsibility. The loss of data at a local database system may affect the MDMS. We can safely assume that the local DBMS can recover from a media failure without user intervention. As the MDMS is also seen by the local DBMS as a user, intervention from the MDMS would also not be required.

The global log must be backed up to make provision for media failure. The procedures for this are well established in centralized databases and can be adapted for MDMS systems [Bar90].

6.1.4 Network failures

Multidatabases depend on the ability of all sites to communicate reliably in order to operate successfully. Most networks today are very reliable with correctness being guaranteed by the underlying protocols. However, failures still occur and a failure can result in a network becoming partitioned into two or more subnetworks. If agents of the same global transaction are active in two different partitions of the network, this could cause a violation in the atomicity of the transaction if agents in one partition decide to commit and agents in the other partition decide to abort and rollback. In general it is not possible to design a non-blocking atomic protocol for an arbitrarily partitioned network. Recovery methods in the case of a network partitioning due to network failure can be either optimistic or pessimistic [Bel92].

- *Optimistic commit protocols*: These choose availability at the expense of consistency and allow updates to proceed independently in the various partitions. On recovery, when the networks are re-connected, inconsistencies are likely. The user will have to assist the ensuing recovery process because the recovery manager will not be able to determine the inconsistencies on its own. On discovering that there is an inconsistency, the system has three choices:

  1. Undo one (or more) of the offending transactions — this could have a cascading effect.
  2. Apply a compensating transaction which involves undoing one of the transactions and notifying any affected external agent that the correction has been made.

3. Apply a correcting transaction, which involves correcting the database to reflect all the updates.

- *Pessimistic merge protocols*: These choose consistency over availability. Here the updates are confined to a single distinguished partition. Recovery is much more straightforward here because on reconnection the updates are simply propagated to all other applicable sites.

6.1.5 DBMS failures

This happens when a condition occurs which stops execution of a DBMS. The failure of a DBMS does not cause a multidatabase system failure since other DBMSs may continue to function.

Recovery from this type of failure is based on information stored on secondary, stable storage. The information thus stored is referred to as a *log*. All transaction actions are logged when they read or write a data item, or when they begin or terminate execution [Bar90].

Two techniques are available for DBMS recovery along with some hybrids of these techniques [Ber87]. The first approach is to *redo* all operations of a committed transaction which have not been recorded in the database at the time of failure. The other option is to *undo* operations effected on the database but whose transaction has not committed. Recovery management algorithms have been defined for both forms of recovery. Specific details are covered in [Ber87, Özs91].

Database system failures in a MDB can be due to software or hardware errors. Software failures can occur in three ways [Bar90]:

- *One of the local DBMSs can fail*: Because the local database systems are autonomous, they are capable of managing all submitted transactions independently. This means that when it fails, it is capable of recovering only committed transactions. Once recovered, the DBMS must be able to notify the transaction's submitter of the termination condition of the transaction. The MDMS will have too ensure consistency of the global transaction after a local site failure by either redoing or undoing subtransactions.

- *The MDMS can fail*: Failure of the MDMS is different. Recovery is required so that the effects of committed global transactions are reflected in each DBMS. The MDMS recovers using global subtransactions since it does not manage local data directly. Details are given in later sections.

- *The MDMS and one of the local DBMSs can fail simultaneously*: Local site DBMS and global site MDMS failures require that all failed systems be restored. Local systems are autonomous and can be recovered independently. Afterwards, the MDMS resumes operation and follows procedures to bring the multidatabase to a consistent state.

The failure of both local and global DBMSs is no more complicated than the failure of either [Bar90].

6.1.6 System failures

This happens when the underlying operating system stops the DBMS. This is seen as a total system failure and the entire MDMS shuts down. The system must be restored before the DBMS can become operational again. The topic of system failures is beyond the scope of this dissertation. This research deals with failures of the MDMS, so the reason for the failure is independent of their management. We will therefore assume that the operating system functions correctly.

6.1.7 Failures to be considered

To ensure atomicity and durability of global transactions, only the following types of failures are the types of failures which need different handling in a multidatabase situation [Geo90]:

- *Subtransaction failures* — These occur when a subtransaction of a global transaction is unilaterally aborted by the local DBMS. This may happen if a deadlock situation must be resolved at the local database and the global subtransaction is chosen as the victim by the local deadlock procedure to be aborted to break the cycle. Timestamp concurrency control methods could also cause global subtransactions to be aborted. Other reasons could be overflows, requests made for nonexistent resources or resource limits exceeded.

- *Site failures* — In this case the contents of volatile memory are lost at the local database or the MDB site. Stable storage will survive this type of failure.

The other failures mentioned in the previous section can be tolerated by a MDMS just as in any other database configuration. The fact that it is a MDMS does not cause any unique problems for these types of failures.

In this chapter we will therefore concentrate on only subtransactions and site failures which place unique requirements on a MDMS.

In the case of subtransaction failures, recovery will be dealt with by either retrying, redoing or compensating, as discussed in section 6.2. Site failures, either of the multidatabase site or the local database sites, must have a specific crash recovery procedure as discussed in section 6.6.

6.2 Issues in Multidatabase Recovery

A multidatabase transaction becomes globally committed when it commits at the MDMS. To complete a globally committed multidatabase transaction, the MDMS has to commit all its subtransactions at the local database systems. A subtransaction that belongs to a

globally committed multidatabase transaction becomes locally committed when the MDMS commits it at the LDBS.

Multidatabase recovery cannot rely on the local recovery procedures of the LDBs. If a subtransaction fails it is automatically rolled back by the local recovery manager. The LDB does not know or care that the global transaction which the subtransaction belongs to has committed. The multidatabase system cannot prevent site failures and unilateral aborts of subtransactions by local systems.

On the one hand, a locally committed subtransaction cannot be rolled back because of the autonomy of the local database system. On the other hand, if one subtransaction of a global transaction has aborted and all other subtransactions have committed, we have to keep trying to run that subtransaction until it also commits in order to maintain global database consistency.

The autonomy of local databases thus causes the following problems [Geo90]:

- The local DBMSs cannot distinguish local uncommitted transactions from uncommitted global subtransactions. When a LDB comes up after a site failure, its local recovery procedures roll back all locally uncommitted subtransactions, even if the global transaction they belong to has already committed.
- Global transactions which have a subtransaction which has committed locally at some LDB cannot be rolled back. The MDMS must complete the failed subtransactions at each of the sites where they failed in order to have global commitment of the global transaction.
- MDMS recovery actions at each site constitute new transactions. From the point of view of the LDB, the new recovery transactions have nothing to do with the failed subtransaction that was perhaps aborted or rolled back.

All the regular protocols for recovery in a homogenous distributed database require sites to cooperate. If nodal autonomy is to be maintained then none of these recovery protocols can be used. The problems facing the recovery manager in a multidatabase are the same as the problems facing a recovery manager in a regular distributed database system when partitioning occurs [Bel92].

If all participating local databases provide a prepare-to-commit operation, then the task of ensuring atomicity is fairly simple. However, if this is not the case, then we have three different ways of handling recovery in the case of global subtransaction failure [Bre95]:

1. *Retry* — the entire aborted subtransaction, and not only its write operations, is run again. This approach is used when a global subtransaction of a globally committed global transaction has aborted [Bar91, Kim93, Vei92].
2. *Redo* — the writes of the failed subtransaction are installed by executing a *redo transaction* consisting of all the write operations executed by the subtransaction. This

approach is also used when a global subtransaction of a globally committed global transaction has aborted [Bre92b, Geo91a, Sop91b].

3. *Compensate* — at each site where a subtransaction of a global transaction did commit, a compensating subtransaction is run to semantically undo the effects of the committed subtransaction. This approach is used when a global subtransaction of a globally aborted global transaction has committed [Lev91a, Per91, Kor90, Nod94].

We will discuss these approaches in more detail in the following sections.

6.2.1 The retry approach

The retry approach will simply re-submit the entire subtransaction to the site at which it failed [Mut91]. If a global subtransaction fails, it is not a trivial matter to simply re-submit the subtransaction to the LDB. [Bar90] proposes a recovery protocol which requires that the MDMS have exclusive access to a local database of a local site during the local recovery procedure in the case of a site failure. However, the re-submission process could violate multidatabase consistency, even in a case where no local transactions are executing.

Example 6.1 : Problem of recovery in a multidatabase

Consider a multidatabase with three local databases: LDB^1, LDB^2 and LDB^3. We also have a global transaction GT_i that reads and writes data items a and b stored in LDB^1 and LDB^2, respectively:

$GT_i : r_{GT}(a),\ r_{GT}(b),\ a = a + b,\ b = b + a,\ w_{GT}(a),\ w_{GT}(b)$

The multidatabase now generates the following subtransactions:

$GST_i^1 : r_{GT_i}(a)$, [wait to receive b], $a = a + b$, [send a], $w_{GT_i}(a)$ at LDB^1

$GST_i^2 : r_{GT_i}(b)$, [send b], [wait to receive a], $b = a + b$, $w_{GT_i}(b)$ at LDB^2

GT_i globally commits and the MDMS commits GST_i^2 at LDB^2, but LDB^1 fails before the local commitment of GST_i^1. If a and b had initial values of 5 and 10 respectively, the value of b becomes 25 but the value of a remains 5. Now the MDMS recovery re-submits GST_i^1. The original value of b is no longer available so that re-submission of GST_i^1 will produce an incorrect value of a. ◇

6.2.1.1 Requirements for retrying a subtransaction

We see from the example above that in order to retry a subtransaction, there should be no data dependencies between GST_i^1 and any other subtransaction of GT_i. Furthermore, the

subtransaction must be *retriable*, that is, if GST_i^1 is retried a sufficient number of times from any database state, it will eventually commit. This is important since before the subtransaction is retried the state of the local DBMS may be changed due to the execution of other local transactions. This should not result in the situation where the subtransaction cannot be committed. It can easily be shown that not every transaction has this property. For instance, if a transaction needs to debit an account and a local transaction empties the account before the transaction is retried, then the transaction cannot succeed. Because of this type of problem, the retry approach is limited [Geo90].

6.2.2 The redo approach

Since the re-submission of subtransactions causes inconsistency, the other approach is to redo all failed subtransactions that belong to globally committed multidatabase transactions. If this approach were used in the above example, the write-ahead log would have stored the values of a and b produced by the subtransactions GST_i^1 and GST_i^2. Then when the LDB comes up after a failure, the recovery procedure would be able to determine the correct value of a and redo by issuing a transaction that simply writes the value of 15 onto a.

A *redo transaction* thus consists of all the writes performed by the subtransaction, and is sent to the local DBMS for execution. If we have a server at the local site, the server has to maintain a *server log* in which it logs the updates of the global subtransactions. If the redo transaction fails, it is repeatedly resubmitted by the server until it commits. Since the redo only consists of write operations, it cannot logically fail.

In Example 4.3 in Chapter 4, we illustrated that in the presence of failures, the local schedule, while serializable from the point of view of the local DBMS, may not be serializable from the point of view of the GTM [Bre95].

Care must therefore be taken to ensure that in the case of a redo approach that the redo transaction will leave the MDB in a consistent state.

To ensure global serializability, Breitbart *et al* [Bre95] state that we need to use additional mechanisms like restricting access to certain data items by local and global transactions or employing a concurrency control scheme which is failure resilient [Bre95]. This aspect was covered in Chapter 4.

6.2.3 The compensate approach

Consider once again the example in section 6.2.1 where transaction GT_i is committed at site LDB^2 and aborted at site LDB^1, and assume that the retry approach is not applicable. We then have to *compensate* for the committed subtransaction GST_i^1. This can be done by issuing a *compensating transaction* CT^1 at site LDB^1 that undoes what GST_i^1 did. For instance, if GST_i^1 reserved a seat for a flight, then CT^1 cancels the reservation. Since the effects of the transaction may have affected the execution of other local transactions, the

resulting state may not be the same as if GST_i^1 had never executed but will be semantically equivalent to it.

We can illustrate this with the following example: Say a transaction reserved the last possible seat on a certain flight. Before the compensating transaction has a chance to reverse the cancellation, another passenger tries to reserve a seat on the same flight, and is turned away. So, while the database is semantically returned to its previous status, the resulting state differs from the original state. Thus, executing compensating transactions does not result in standard atomicity of transactions. The resulting notion of atomicity is referred to as *semantic atomicity.*

In [Gar87], Garcia-Molina and Salem use the term *saga* to refer to a collection of semantically atomic subtransactions. To ensure semantic atomicity, the GTM must keep a log of all the GST_i subtransactions that have been committed.

A compensating transaction, besides performing an inverse of the function performed by GT_i, must also ensure that after it commits, the global constraints between different local sites where GT_i executes, hold. Even though the execution of a compensating transaction CT will re-establish the consistency constraint violated due to the partial commitment of a global transaction, it will not prevent other transactions that execute at these local sites before CT executes from seeing inconsistent data. This problem has been studied by Levy, Korth & Silberschatz and Mehrotra, Rastogi, Korth & Silberschatz [Lev91b, Meh92d], and two different protocols were proposed that guarantee strong correctness in the presence of a combination of global and compensating transactions.

The design of compensating transactions has been discussed in the literature [Gar83]. Some subtransactions may not have simple compensations. For example, if a subtransaction deposits funds in an account and those funds are withdrawn before the compensating transaction can be run, then compensation cannot take place. Some other transactions are not compensatable; e.g. firing a missile [Bre95].

6.3 Extension of the Database Model

Barker & Özsu [Bar91] define various concepts and terminology to reason about multidatabase recovery. In Chapter 3 we defined the concepts of: recoverable, avoids cascading aborts and strict. We will now extend this in order to use these concepts in a multidatabase environment. We will define three levels of recoverable local histories: *local recoverable, avoids local cascading aborts,* and *locally strict.*

Definition 6.1 — *Local recoverable (LRC)*

A local history LH^k is *local recoverable* (LRC) if, every transaction that *commits* reads only from committed transactions .

[Bar91] □

Definition 6.2 — *Avoids local cascading aborts (ALCA)*

A local history LH^k *avoids local cascading aborts* (ALCA) if all transactions read from committed transactions .

[Bar91] □

Definition 6.3 — *Locally strict (LST)*

A local history LH^k is *locally strict* (LST) if :

1. it avoids cascading aborts, and
2. whenever $w_j(X) \prec O_i(X)(i \neq j)$ then $N_j \prec O_i(X)$ where $N_j \in \{a_j, c_j\}$ and $O_i(X)$ is $r_i(X)$ or $w_i(X)$.

[Bar91] □

These concepts are best illustrated by means of an example:

Example 6.2 : Different levels of recoverability

Different levels of recoverability at the local DBMSs are illustrated by a sequence of increasingly restrictive histories. Assume the following history is produced by a local DBMS scheduler:

$$LH^1 : r_1^1(d);\ r_1^1(e);\ w_1^1(d);\ \hat{r}_1^1(e);\ r_2^1(d);\ \hat{w}_1^1(e);\ \hat{w}_1^1(d);\ w_2^1(d);\ \hat{c}_1^1;\ c_2^1;\ c_1^1 \quad \text{(H1)}$$

H1 is not local recoverable because c_2^1 precedes c_1^1. This is a problem because according to the definition if a transaction commits and it reads from the results of another transaction, then that transaction must commit first. This problem is corrected as follows:

$$LH^1 : r_1^1(d);\ r_1^1(e);\ w_1^1(d);\ \hat{r}_1^1(e);\ r_2^1(d);\ \hat{w}_1^1(e);\ \hat{w}_1^1(d);\ w_2^1(d);\ \hat{c}_1^1;\ c_1^1;\ c_2^1 \quad \text{(H2)}$$

The history H2 is not ALCA because $w_1^1(d) \prec r_2^1(d) \prec c_1^1$ so GST_2^1 reads from GST_1^1 before GST_2^1 commits. The following history is ALCA:

$$LH^1 : r_1^1(d);\ r_1^1(e);\ w_1^1(d);\ \hat{r}_1^1(e);\ c_1^1;\ r_2^1(d);\ \hat{w}_1^1(e);\ \hat{w}_1^1(d);\ w_2^1(d);\ \hat{c}_1^1;\ c_2^1 \quad \text{(H3)}$$

H3 is not locally strict since $\hat{w}_1^1(d) \prec w_2^1 \prec \hat{c}_1^1$. The local history:

$$LH^1 : r_1^1(d);\ r_1^1(e);\ w_1^1(d);\ \hat{r}_1^1(e);\ c_1^1;\ r_2^1(d);\ \hat{w}_1^1(e);\ \hat{w}_1^1(d);\ \hat{c}_1^1;\ w_2^1(d);\ c_2^1 \quad \text{(H4)}$$

is LST since LT_1^1 commits before GST_2^1 updates the value of d.

[Bar91] ◇

Hadzilacos [Had88] has proved that $LST \subset ALCA \subset LRC$ holds between these histories. These concepts can be extended to the multidatabase environment by applying them to global histories as shown in the following section [Bar91].

6.4 Recoverability in Multidatabases

We need to determine the conditions under which multidatabase consistency can be preserved in the event of a failure. In chapter 3 we showed that to ensure correctness of a local database in case of failures, schedules must at least be recoverable.

We also defined the recoverability concept with respect to centralized databases in Chapter 3. [Geo90] states that recoverability is the weakest possible requirement that guarantees correctness in a DBMS in the presence of failures.

6.4.1 The problem of multidatabase recoverability

Global recoverability requires that two conditions be met [Bar91]:

1. All local histories are local recoverable.
2. All global subtransactions submitted on behalf of a global transaction have the same termination condition.

We can formalize this requirement:

Definition 6.4 — *Global recoverability*

A global history is *globally recoverable (GRC)* if,

1. all LH^i are at least LRC, and
2. all global transactions terminate uniformly .

[Bar91]

□

The first condition requires that local histories be at least locally recoverable. For the purpose of GRC, we are interested in the set of GSHs which comprise the global history. If an arbitrary local history LH^k is locally recoverable it follows that any subset of LH^k exhibits the same property. This is the only way that we can guarantee that global subtransaction histories are recoverable given the autonomy of the LDBs. Since $GSH^k \subseteq LH^k$ it follows that GSH^k is also LRC. This could mean that more restrictive histories could occur in a

GRC history. For example, a MDMS which guarantees LRC at LDB^1, ALCA at LDB^2 and LST at LDB^3 could provide GRC histories.

The second condition provides consistency across DBMS boundaries and is summarized by the following definition:

Definition 6.5 — *Global transaction termination uniformity*

A global transaction (GT_i) *terminates uniformly* if either one of the following conditions hold:

1. if $\exists GST_i^l \in \mathcal{GST}^l$ where $a_i^l \in GSH^l$, then $\forall GST_i^k$ where the write set of GST_i^k is not empty, $\nexists c_i^k \in GSH^k$.
2. if $\exists GST_i^l \in \mathcal{GST}^l$ where $c_i^l \in GSH^l$, then $\forall GST_i^k$ where the write set of GST_i^k is not empty, $\nexists a_i^k \in GSH^k$.

[Bar91]

□

This definition means that if any global subtransaction has aborted at any DBMS, then all other global subtransactions belonging to that global transaction have either aborted or they have not yet terminated. On the other hand, if any global subtransaction has committed at any DBMS, then all other global subtransactions have either committed or are still active. The restriction to the write-set is imposed because read-only transactions do not affect databases and we therefore ignore them when considering reliability.

Since avoidance of cascading aborts and strictness are subsets of recoverable histories they also need to be defined:

Definition 6.6 — *Avoids global cascading aborts (AGCA)*

A global history avoids global cascading aborts (AGCA) if,

1. all LH^i are at least $ALCA$, and
2. all global transactions terminate uniformly .

[Bar91] □

Definition 6.7 — *Globally strict (GlST)*

A global history is globally strict (GlST) if,

1. all LH^i are at least LST, and
2. all global transactions terminate uniformly .

[Bar91] □

The major difference between global recoverable histories and local recoverable histories is that all global subtransactions, for any global transaction, terminate the same way. This is best illustrated by an example:

Example 6.3 *: Global recoverability*

Consider the following histories:

$LH^1 : r_1^1(d); r_1^1(e); w_1^1(d); \hat{r}_1^1(e); r_2^1(d); \hat{w}_1^1(e); \hat{w}_1^1(d); w_2^1(d); \hat{c}_1^1; c_2^1; c_1^1;$

$LH^2 : r_2^2(u); w_2^2(s); w_1^2(s); \hat{r}_1^2(u); \hat{w}_1^2(u); \hat{c}_1^2; c_1^2; a_2^2;$

Both LH^1 and LH^2 are LRC but the global history is not GRC. Consider the following projection of these histories:

$GSH^1 : r_1^1(d); r_1^1(e); w_1^1(d); r_2^1(d); w_2^1(d); c_2^1; c_1^1;$

$GSH^2 : r_2^2(u); w_2^2(s); w_1^2(s); c_1^2; a_2^2;$

Since GST_2^1 commits in GSH^1 but GST_2^2 aborts in GSH^2 it is evident that $GH = GSH^1 \cup GSH^2$ is not GRC. If LH^2 was executed as follows:

$GSH^2 : r_2^2(u); w_2^2(s); w_1^2(s); c_1^2; c_2^2;$

the GH would be GRC. Similar arguments could be made for AGCA and GIST [Bar91]. ◇

We can now conclude that $GlST \subset AGCA \subset GRC$ which follows from the previous definitions and the established relationship between local histories [Bre95, Bar91].

In a multidatabase system, for each subtransaction interrupted by failure, a corresponding recovery operation must be issued. From the point of view of the LDB, recovery transactions have no connection to the transaction that they are intended to complete. Because of this, if multidatabase recovery and local transactions interleave, consistency is difficult to preserve. To illustrate this problem, consider the following example:

Example 6.4 *: Recovery transactions*

We have two local systems LDB^1 and LDB^2. Data item a is stored at LDB^1 and data item b is stored at LDB^2. Consider the global transaction that accesses data items at these two databases:

$GT_i : r_{GT_i}(a), w_{GT_i}(a), r_{GT_i}(b), w_{GT_i}(b)$

Suppose that GT_i is globally committed but a site failure occurs before local commitment of the subtransaction GST_i^1 of GT_i at LDB^1. As far as LDB^1 is concerned, GST_i^1 is not locally committed. Therefore, during recovery when it comes up again, it rolls back GST_i^1. Next, the MDMS realizes that GT_i was globally committed and issues a recovery transaction R to redo GST_i^1. But, just after LDB^1 comes up, the following local transaction is executed before the recovery transaction R.

$T : r_T(a), w_T(a)$

This results in the following local schedule at LDB^1:

$r_{GT_i}(a)$, [GT_i is aborted by local recovery here], $r_T(a), w_T(a), c_T, w_R(a), c_R$

The above schedule is serializable and strict and it is allowed by the rigorous LDB^1. However, R uses the value of a which is read by GT_i and is recorded in the multidatabase log. So in the view of the MDMS, $w_R(a)$ is logically performed by GT_i. Therefore the execution is logically equivalent to the following globally non-serializable schedule:

$r_{GT_i}(a), r_T(a), w_T(a), c_T, w_{GT_i}(a), c_{GT_i}$

[Geo91a] ◇

6.5 Global Logging

Critical stages in a global subtransaction's execution must be logged to ensure consistency in the presence of failures. The global scheduler transmits three types of information to the global recovery manager [Bar91]:

1. *A global transaction initiation message* that identifies the global transaction and provides a list of global subtransactions to be processed.
2. The *global subtransactions* are sent to the recovery manager where they are logged before being sent out to the appropriate LDB.
3. Each *global transaction's termination condition* is recorded to ensure that only committed global transaction affect the MDMS. The global recovery manager informs the global scheduler of the termination condition, at which time the GT is committed.

Four critical points must be logged on stable storage [Bar91]:

1. Termination conditions of all global transactions.

2. Each GST which has completed must be recorded and its termination condition must be logged.

3. When a GST has been submitted to a LDB, it is recorded on an active list in the log so that the global restart operation knows which GSTs may be outstanding.

4. When a GST completes, information may be returned in the form of a result.

This information must be saved so that if a failure occurs it is available and recovery can be done.

6.6 Multidatabase Recovery Approaches

In this section we will take a look at how the core group handles the recovery question. As recovery and global commitment go hand in hand one will often find the recovery issue merging with the global commitment issue.

6.6.1 Barker & Özsu's basic MDB model

Barker & Özsu's transaction model is described in synopsis 4.22. Barker & Özsu [Bar91] propose a recovery protocol which does not require modification of DBMS code, but assumes that all LDBs are willing to cooperate with the MDMS in the recovery process. By cooperation is meant that each LDB recovers in conjunction with the MDMS. This model assumes that all local database systems provide strict schedules.

- **Multidatabase recovery from a LDB site failure:**

  When a local database system comes up after a failure, it typically is started, recovery is performed and the users are permitted to access the database once more. Barker & Özsu propose a modified restart process [Bar91]:

  1. Restart the DBMS.
  2. Recover the database using information in the local log.
  3. Open the database so that the MDMS has exclusive access.
  4. Establish a handshake with the MDMS to notify it that the local site has recovered. Wait for a response. This type of facility is already provided by ORACLE[1], Sybase and INGRES[2].
  5. The MDMS submits all GSTs that were ready to commit at the time of failure. This can be done because of the strictness requirement.

[1]ORACLE is a registered trademark of Oracle Corporation

[2]INGRES is a registered trademark of Relational Technology

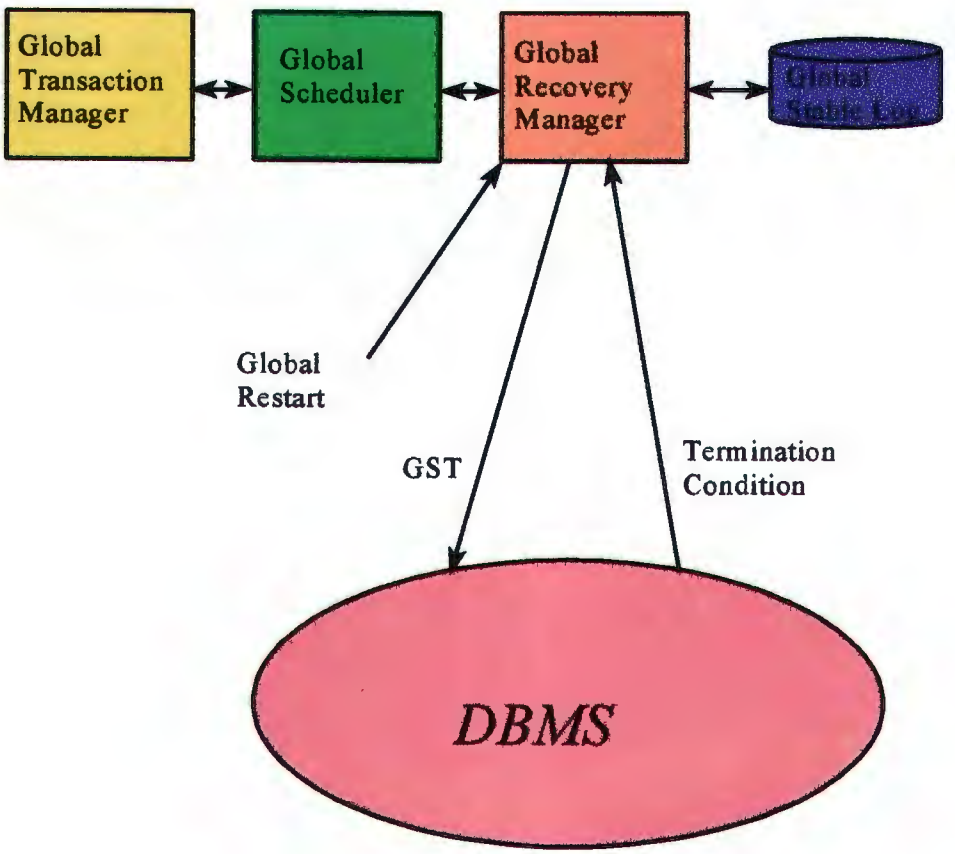


Figure 6.1: Barker & Özsu's global recovery manager architecture

[Bar90, p98]

6. The MDMS notifies the database administrator that exclusive access is no longer required.
7. The DBA opens the database to normal user access.
8. Terminate the local restart process.

During the restart process only the database administrator is permitted to access the database until after the recovery has completed in step 2. The opening of the database in step 3 does not require the database administrator to allow access to all possible users.

- **Multidatabase recovery from a MDMS site failure:**

  Figure 6.1 illustrates the architecture of the global recovery manager.

  The global recovery manager requires a global stable storage. In the event of a system failure, all the necessary information must be available to facilitate the recovery

process. If the MDMS fails, a global restart operation is issued which re-establishes the status of outstanding global transactions and global subtransactions. Since the MDMS does not manipulate data, it only has to ensure that all global transactions and global subtransactions complete correctly [Bar90].

The following steps are performed in the case of a MDMS site failure:

1. The DBA restarts the MDMS, it continues to accept responses from active global subtransactions but accepts no new global transactions.
2. The MDMS determines what happened during the failure. This involves the submission of requests to each DBMS to determine the current status of global subtransactions active at the time of failure. The active global subtransactions can be in several states:
   - the local site could also have failed in which case it must first be restarted and then the status of all subtransactions at that site must be checked.
   - the global subtransaction is still active at a site in which case it is allowed to continue.
   - the global subtransaction could have committed while the MDMS was down. In this case, the completion is recorded in the global log.
   - the global subtransaction could have aborted while the MDMS was down in which case this fact must be recorded in the global log.

   The subtransactions of committed global transactions are at least ready to commit and are therefore handled as follows:
   - if the global subtransaction was aborted at the local site it is resubmitted to the local DBMS.
   - if the global subtransaction committed, nothing is done.
   - if the global subtransaction does not appear in the log the transaction is still active so it is allowed to complete.

   The subtransactions of aborted global transactions are handled as follows:
   - if the global subtransaction was ready, communicate the abort.
   - if the global subtransaction is active, when it becomes ready, abort it.
   - if the global subtransaction is aborted, do nothing.

Barker [Bar90] proves that a failed MDMS can recover using the global restart process as described above. Tung [Tun92] also presents a protocol which requires exclusive access during recovery and also assumes that local sites use 2PL.

6.6.2 Pu's hierarchy of superdatabases

Pu [Pu88] states that since the local database systems are heterogeneous, it is necessary that each element database maintains the undo/redo information locally. Since the super-

database stores only the global information, it relies on the element databases for local recovery.

The superdatabase is the coordinator for the element databases during the commit process so it must record the transaction on stable storage. Otherwise, a crash during the uncertain period would hold resources in the element databases indefinitely.

Pu states that the best method for superdatabase recovery is logging. No before or after images have to be saved so versions are not viable. The superdatabase log is separate from the element database logs. For each transaction, the superdatabase log should have the following information:

- Participant subtransactions.
- Parent superdatabase.
- Transaction state (prepared, committed, or aborted).

The superdatabase has to remember subtransactions because if the superdatabase crashes the transaction does not necessarily abort. If the superdatabase restarts quickly enough, it may still be able to allow the subtransactions of a transaction to commit. If a transaction was in the active state when the superdatabase crashed, the superdatabase simply waits for retransmission of the two-phase commit from the parent. If it is the root, it restarts the two-phase commit. If a transaction was in the prepare state when the superdatabase crashed, the superdatabase inquires from the parent about the outcome of the transaction. If the transaction committed, the results are retransmitted to all the subtransactions.

6.6.3 Breitbart *et al*'s work

Breitbart *et al*'s [Bre90a] approach to recovery prevents anomalies caused by global transactions after failure by maintaining global locks at the multidatabase level in order to coordinate execution of global transactions. An operation of a global transaction must first obtain a lock on a data item before it can be submitted to the local database system.

Breitbart *et al*'s model partitions the data into locally and globally updateable data. In order to be able to recover from failures and leave the database in a consistent state, they impose further restrictions. They require all transactions modifying globally updateable transactions should not be allowed to read locally updateable data items. This restriction is called the *global consistency requirement.* If a global transaction is read-only, then it may read both sets of data. The algorithm also uses a commit graph to control the commit order of global transactions.

6.6.4 Elmagarmid *et al*'s work

The recovery aspect is not addressed in Elmagarmid *et al*'s model [Elm87].

6.6.5 Chen *et al*'s distributed MDMS

Bukhres *et al* [Buk93] describe the implementation of the transaction management scheme set out in [Che93]. The implementation, InterBase, has five components that can fail: the GTMs, the interfaces, the servers, local systems and the local site. Among them, only the interfaces run continuously. An interface failure affects the local site on which it runs while GTM and server failure affects only a single global transaction or global subtransaction respectively. A local system crash may affect various global transactions but its autonomy limits our ability to restore it to its precrash state. We outline the crash recovery protocol as follows:

- *GTM crash-recovery protocol*: If an interface detects that a GTM has failed and it has not yet created a server for the global transaction which is coincident with that GTM, it simply arranges the relative execution order by removing the transaction from the order. If the server has already been created, it is informed by the interface of the crash and it then aborts itself.

- *Interface crash-recovery protocol*: A GTM or a interface server detects failure of an interface first and attempts to reactivate it. Each interface maintains a write-ahead log that allows it to recover to a precrash state after being reactivated. If duplicate interfaces have been started by various servers, one of them will terminate after negotiation.

- *Service crash-recovery protocol*: If a GTM detects that a server has crashed, it requests the interface to create a replacement. The interface complies if doing so does not violate the execution order and if the crash is not due to the crash of the local system. If the replacement cannot be created, the GTM may abort itself and other servers.

- *Local system crash-recovery protocol*: If a server detects the crash of its associated local system, it attempts to reactivate it, otherwise it aborts the subtransaction for which it was created and reports the problem to the interface and the GTM. The GTM then tries to reactivate the local system and if that also fails it aborts the global transaction and itself.

Chen *et al* do not address the issue of total site failure.

6.6.6 Kang & Keefe's decentralized GTMs

Kang & Keefe's [Kan93] GTMs are distributed so there is no MDMS site *per se*. In this model we therefore only have to consider a local site failure. If the local site fails, the procedure to recover a subtransaction at a participant site will be:

- if the subtransaction was active, do nothing because when the coordinator sends the prepare message out it will get no response and the global transaction will be aborted.

- if the subtransaction was in the prepared to commit state, the local GTM must redo the subtransaction from the information in the log. This could, of course cause serializability problems.

The situation if the coordinator site fails is not addressed, but presumably the participants would wait for the prepare message from the coordinator site and timeout after a while and abort their subtransaction.

6.6.7 Garcia-Molina *et al*'s sagas

Garcia-Molina *et al* [Gar87] propose two options for dealing with failure — backward recovery (compensate for executed transactions) or forward recovery (execute the missing transactions).

- **Backward recovery**: For this type of recovery the system needs compensating transactions. If the GTM receives an abort-saga command, it initiates backward recovery. The GTM will record the instruction to the log and then orders compensating transactions to undo what the saga did.

  It would also be able to recover from crashes. After a crash, the GTM would determine the status of all sagas. If a saga has begin-saga and end-saga entries in the log, then the saga completed and no further action is necessary. If there is a missing end-saga, then the saga did not complete and will be aborted. The GTM will then attempt to determine which transaction the saga belonged to and compensate all other sagas of that global transaction in order to maintain consistency [Gar87].

- **Forward recovery**: For this type of recovery the system needs save-points as well as a reliable copy of the code for all missing transactions. A save point is a place in a saga where the system is forced to save the state of the running application program. The save point to be used may be specified by the system or the application depending on which aborted the saga. The save-points reduce the amount of work after a crash: instead of trying to recover all outstanding transactions, the system only needs to recover for transactions executed after the last save-point. In the case of a system crash, the most recent save-point can be identified for each active saga.

  After every crash, the recovery manager will abort the last executing transaction, and start the saga at the point where this transaction had started.

6.6.8 Yoo & Kim's client server approach

Yoo and Kim [Yoo95] make use of a commit protocol with state changes as shown in the state diagram in Figure 5.1. The actions that the MDBS_TM and the agent can take in case of failure can be considered:

- **Recovery from MDBS_TM site failures:** The following cases are possible:

  1. *MDBS_TM site fails while in the initial state.* This is before the MDBS_TM has initiated the commit procedure. Therefore, it will write an *abort-record* in the log upon recovery and send an *global-abort* message to all the participating LDBs.
  2. *MDBS_TM site fails while in the wait state.* In this case, the MDBS_TM has sent the prepare command. Upon recovery, it will restart the commit procedure for this transaction from the beginning by sending the prepare message one more time.
  3. *MDBS_TM site fails while in the commit or abort states.* In this case, the MDBS_TM will have informed the agents of its decision and terminated the transaction. Thus, upon recovery, it does not need to do anything if all acknowledgements have been received. Otherwise, the termination protocol for MDBS_TM is started.

- **Recovery from agent site failures:** There are three alternatives:

  1. *An agent site fails while in the initial state.* Upon recovery, the LDBMS rolls back the effects of all the uncommitted transactions. Therefore, the subtransaction has also aborted. At this time, the agent has no information about the subtransaction, and thus no action is required (in the case of a LDBMS failure, the agent releases all the sub-level locks held in memory). This causes the MDBS_TM to timeout, and eventually to invoke MDBS_TM's terminating procedures.
  2. *An agent site fails while in the local commit state.* In this case, the LDBMS has committed the subtransaction. Upon recovery, the agent at first initializes the in-memory stub level locking table and then treats this failure as a timeout in the local commit state.
  3. *An agent site fails while in the global commit or abort states.* These states represent the termination conditions. So, upon recovery, the agent does not need to take any special action.

There is no recovery required in this protocol for subtransaction failures since this protocol has no prepared state. Even one failure in local commit operation will assure that the transaction is globally aborted, which results in a globally consistent state.

6.6.9 Other relevant research

6.6.9.1 Georgakopoulos's work

Full details of Georgakopoulos's recovery method can be found in [Geo90]. His approach has the following distinguishing features:

- The LDBs only have to guarantee serializability and strictness.

- Local and global transactions are allowed to read and update the same data items.
- The proposed MDMS recovery scheme takes advantage of the local DBMSs by minimizing replication of recovery tasks.
- It is assumed that a transaction cannot be aborted by the local DBMS after all its operations have been completed.
- The recovery process has exclusive use of the local DBMS during recovery from a site failure.

It has been proved by Hwang & Srivastava that Georgakopoulos's algorithm, used in conjunction with a multidatabase concurrency control algorithm, achieves global serializability in the presence of failure [Hwa94].

6.7 Analysis

The subject of crash recovery has not been given much attention in the literature but we considered the various different approaches represented in our core group.

Barker & Özsu [Bar91] assume an exclusive access period after failure, as does Georgakopoulos [Geo91a]. Pu [Pu88] assumes that the root superdatabase and "leaves" cooperate in the recovery process. Breitbart *et al* [Bre90a] partition data in order to facilitate database consistency in the face of failures. Chen *et al* [Che93] exploit the recovery procedures of the underlying local DBMSs without violating the local autonomy as is done in varying degrees in Barker *et al*, Pu and Breitbart *et al*. Chen *et al* do not deal with the question of a total site failure but no doubt the protocol could be extended to handle that contingency. Kang & Keefe [Kan93] maintain global reliability by assuming cascadeless schedules at the local sites and partitioning data as is done in Breitbart *et al* [Bre90a].

Garcia-Molina *et al* [Gar87] need either compensating transactions or a system of savepoints in the transactions in order to recover from failures. Yoo *et al*'s scheme [Yoo95] also maintains maximal local autonomy and recovers by using the logs and the locking information. It should be borne in mind that all transactions in this model are routed via the stub so that there will be no problem after startup with local transactions seeing inconsistent data because the stub will simply delay local transactions until the recovery procedures have been run.

6.8 Summary

This chapter addressed two issues, recovery and recoverability. The research undertaken has shown that efficient recovery is needed to ensure reliability of the multidatabase system and that the recovery question is not trivial in multidatabase systems. The types of failures which can occur in a multidatabase system have been discussed and the failures which need

unique handling were identified. The various approaches to recovery were discussed and illustrated by means of examples. The issue of recoverability was addressed by extending the database model to include concepts relating to the recoverability of schedules in a multidatabase. The various recovery procedures used by the core group were examined and an analysis of how they measure up was given. Crash recovery would seem to be a weak point in many multidatabase systems and research into better ways to effect recovery without violating local autonomy still remains to be done.

Chapter 7

Appraisal

We can now evaluate and comment on the schemes presented in each of the core group approaches. We identified the autonomy dimension as being the most important dimension when we consider multidatabase architectures. In the first section of this chapter we will use the quantification method as outlined in Section 2.1.4 to work out an autonomy violation for each of the core group schemes. Thereafter, a multidatabase transaction processing model will be proposed, together with the concurrency control, recovery and reliability scheme which should be used for the transaction management scheme.

7.1 Autonomy Quantification

7.1.1 Barker & Özsu's basic MDB model

Barker & Özsu [Bar91] state that since their scheme emulates 2PC and forces no ordering on the local DBMS, it affords greater autonomy, but it also states that each DBMS must guarantee a strict level of service which according to Mullen *et al* [Mul92] definitely violates local autonomy. According to Mullen *et al* [Mul92], the only assumption that should be made about the local DBMSs is that they support the ACIDity properties of their transactions. Although it may be unrealistic to expect a level of strictness in all local members of multidatabase systems, we have seen that one must compromise somewhere in order to implement concurrency transaction management and this method compromises on local autonomy. The autonomy quantification for this scheme is:

```
Modification Dimension
    System              0.75        Reliability protocol requires strictness
    Data                0           Data remains untouched
    Design              X           Not discussed
Total value: 0.75
```

```
Execution Dimension
    Local Transaction  0          Execute as usual
    Global Transaction 0.5        Handshake required at commit point
Total value: 0.6124

Information Exchange Dimension
    Execution          0.25       Failures require MDMS to query DBMS
    Data               X          Not discussed
    Schema             0          External Schema only
Total value: 0.25

The overall autonomy violation is 1.0.
```

7.1.2 Pu's hierarchy of superdatabases

Pu [Pu88] does not make provision for failures and so is a scheme that perhaps has a lot of scope for further research. Pu's superdatabase approach has four good characteristics [Pu88]:

- Superdatabases guarantee the atomicity of global updates across the element databases. This includes both reliability atomicity as well as concurrency atomicity.
- The design of superdatabases is adaptable to a variety of crash recovery methods and concurrency control methods used in the element databases. This protocol assumes that there must be agreement in order to commit but the protocol is independent of local crash recovery procedures used to undo and redo local transactions at the local databases.
- Databases built with superdatabases are extensible by construction. Element databases can be added or removed without changing the superdatabase.
- Transactions local to element databases run independently of the superdatabase, which intervenes only when needed for synchronization or recovery of supertransactions across different element databases.

Pu's transaction management protocol was implemented on a prototype called Harmony at Columbia University [Pu91b]. Global deadlock detection and resolution are areas that still have to be researched for this model and of course many multidatabase applications may find the local autonomy violation unacceptable. The autonomy quantification for this scheme is:

```
Modification Dimension
    System              0.75        Inferred in discussion on reliability
    Data                0           Data remains untouched
    Design              X           Not discussed
Total value: 0.75

Execution Dimension
    Local Transaction   0.5         Serial orderings sent to MDMS
    Global Transaction  0.5         Serial orderings sent to MDMS
Total value: 0.866

Information Exchange Dimension
    Execution           0.5         MDMS queries DBMS for commit order
    Data                X           Not discussed
    Schema              0           External Schema only
Total value: 0.5

The overall autonomy violation is 1.25.
```

7.1.3 Breitbart *et al*'s work

Breitbart *et al*'s algorithm can be summarized as follows:

- Each local DBMS enforces use of strict 2PL.
- A global subtransaction which updates globally updateable data cannot read locally updateable data.
- The GTM maintains locks for all globally updateable data.
- A commit graph is used to prevent cyclic commit executions.
- Breitbart *et al*'s algorithm also achieves global serializability [Hwa94].

We will consider the latest research by Breitbart *et al*, namely the server model discussed by Georgakopoulos [Geo91a]. This scheme ensures global database consistency and freedom from global deadlocks. They assume that strict 2PL is used and that data items can be partitioned. Breitbart *et al* claim that the restrictions imposed are administratively easy to maintain. Breitbart *et al* feel that the payoff from the imposed restrictions is significant enough to justify them.

This protocol guarantees global consistency in the face of failures and does not violate local autonomy. The autonomy quantification for this scheme is:

```
Modification Dimension
    System              0.75        Inferred in discussion on reliability
    Data                0.5         Timestamp data item in each database
    Design              X           Not discussed
Total value: 0.9

Execution Dimension
    Local Transaction   0           Execute as always
    Global Transaction  0.5         GST communicates with MDMS at commit
Total value: 0.6124

Information Exchange Dimension
    Execution           X           Not evident from literature
    Data                X           Not discussed
    Schema              0           External Schema only
Total value: 0

The overall autonomy violation is 1.0897.
```

7.1.4 Elmagarmid *et al*'s work

Elmagarmid *et al* present a framework for designing concurrency control control protocols using a top-down approach. This means that the global serialization order of global transactions must be determined at the global level before their being submitted to the local sites. He presents two mechanisms for ensuring global serialization at the local sites. The first controls the submission of global subtransactions by using a stub process and the second controls the execution of global subtransactions by modifying local schedulers [Elm87]. Elmagarmid's approach has the following advantages:

- No global deadlock.
- Simple global control.
- No inter-site communication.
- Fewer global transactions are aborted.

The autonomy quantification for the stub approach is:

```
Modification Dimension
    System              0.75        2PL required
    Data                0.5
    Design              X           Not discussed
Total value: 0.9

Execution Dimension
    Local Transaction   0           Execute as usual
    Global Transaction  0.5         Handshake required at commit point
Total value: 0.6124

Information Exchange Dimension
    Execution           0.25        Failures require MDMS to query DBMS
    Data                X           Not discussed
    Schema              X           Not discussed
Total value: 0.25

The overall autonomy violation in the stub approach is 1.116.
```

Modification of local scheduler approach:

```
Modification Dimension
    System              1           Local DBMS modified
    Data                0.5         Order stamp added
    Design              X           Not discussed
Total value: 1.118

Execution Dimension
    Local Transaction   0.5         Coordination required with MDMS
    Global Transaction  0.5         Communication with MDMS at commit time
Total value: 0.866

Information Exchange Dimension
    Execution           1           Queries MDMS for global serialization
                                        order
    Data                X           Not discussed
    Schema              X           Not discussed
Total value: 1

The overall autonomy violation in the modification of the
local scheduler approach is 1.73.
```

This model was extended by Chen *et al* [Che93] where the reliability and recovery aspects of the model were addressed.

7.1.5 Chen *et al*'s distributed MDMS

This scheme has been implemented in InterBase and has the minimum autonomy violation of all the transaction management schemes in our core group [Che93]. The attractive aspects of this scheme are that it makes no assumptions about the characteristics of the underlying local database systems but rather exploits those characteristics in order to achieve global database consistency. The autonomy quantification for this scheme is:

```
Modification Dimension
    System              0.25        Software above the DBMS
    Data                0.5         Ticket added
    Design              X           Not discussed
Total value: 0.559

Execution Dimension
    Local Transaction   0           Execute as always
    Global Transaction  0.5         GST communicates with MDMS at commit
Total value: 0.6124

Information Exchange Dimension
    Execution           0           No exchange
    Data                X           Not discussed
    Schema              0           External Schema only
Total value: 0

The overall autonomy violation is 0.829.
```

This approach will be discussed in more detail in section 7.2.

7.1.6 Kang & Keefe's distributed GTMs

Kang & Keefe's scheme implements a distributed GTM [Kan93] which is attractive because of the fault tolerance thereof but unfortunately it partitions data items which once again violates local autonomy. As a whole this scheme does not score badly on the autonomy stakes which makes it an attractive alterative to Chen *et al*'s scheme. The autonomy quantification for this scheme is:

```
Modification Dimension
    System              0.75        Reliability protocol requires
                                        cascadeless schedules
    Data                0.5         Tickets and timestamp required
    Design              X           Not discussed
Total value: 0.9

Execution Dimension
    Local Transaction   0           Execute as usual
    Global Transaction  0           Execute as usual
Total value: 0

Information Exchange Dimension
    Execution           0.25        Queries LTM about transaction failures
    Data                X           Not discussed
    Schema              X           Not discussed
Total value: 0.25

The overall autonomy violation is 0.93.
```

7.1.7 Garcia-Molina & Salem's sagas

Garcia Molina & Salem's approach would have limited application in multidatabase environments because of the possible difficulty of breaking up transactions into interleavable pieces. It is also not always possible to design compensating transactions which this model requires. The autonomy quantification for this scheme is:

```
Modification Dimension
    System              0.75        Saga daemon required &
                                        savepoint abilities required
    Data                0.5         Savepoints and log information required
                                        to be written on the database
    Design              X           Not discussed
Total value: 0.9

Execution Dimension
    Local Transaction   0           Execute as usual
    Global Transaction  0           Execute as usual
Total value: 0
```

```
Information Exchange Dimension
    Execution          0.25        Queries LTM about transaction failures
                                         if the facility is available
    Data               X           Not discussed
    Schema             X           Not discussed
Total value: 0.25

The overall autonomy violation is 0.93.
```

7.1.8 Yoo & Kim's client server approach

Yoo & Kim do not discuss the global concurrency control protocol their scheme would use but rather concentrate on a reliable global commit protocol. According to Yoo & Kim [Yoo95], the commit protocol they propose has the following characteristics:

- It preserves execution autonomy because the local DBMS can unilaterally abort any global subtransaction. It preserves communication autonomy because it doesn't need to communicate its control information to the MDMS. It preserves design autonomy because no existing DBMS code is changed.
- There is no restriction on transaction application which is assumed in previous work [Tan93, Vei92].
- The data is not divided up into locally and globally updateable groups as is done by Breitbart *et al* [Bre92b]. Also, data dependency between subtransaction of a global transaction does not incur any problem in this protocol.
- The protocol is failure resistant.

The autonomy quantification for this scheme is:

```
Modification Dimension
    System             1           Changes are made to DBMS procedure calls
    Data               0.5         Stub-level locks
    Design             X           Not discussed
Total value: 1.118

Execution Dimension
    Local Transaction  0.75        Update transactions submitted via the stub
    Global Transaction 0.5         Agent communicates with MDMS at commit
Total value: 0.901
```

```
Information Exchange Dimension
    Execution          0          No exchange
    Data               X          Not discussed
    Schema             X          Not discussed
Total value: 0

The overall autonomy violation is 1.435.
```

7.2 A Multidatabase Transaction Processing Model

After due consideration of the research done in this field, we have decided on the distributed GTM model presented in [Che93] (see section 4.2.5). The reasons for this are the following:

- The GTM is distributed.
- The model is failure resistant.
- The use of the interface allows us considerable leeway in how we handle global transactions. The interface can either send transaction operations to the local DBMS an operation at a time or send through predetermined service requests to the DBMSs at the sites.
- A strong recommendation is that this scheme has been successfully implemented in the InterBase system at Purdue University.
- No assumptions are made about the local sites involved in the multidatabase system. Heterogeneity of local database systems is accommodated easily by the model.
- Local database system autonomy is maintained.
- New database systems are very easily added to and removed from the multidatabase system.

The architectural model is illustrated in Figure 4.5. The GTM in this model does the following [Buk93]:

1. coordinates the concurrent execution of global transactions;
2. interprets the execution of a global transaction;
3. manages the dataflow within a global transaction;
4. ensures the reliable execution of a global transaction;
5. recovers from errors.

Each MDMS transaction is parsed into a set of subtransactions, each of which consists of operations or a service request to an individual LDB. The scheduling order of these subtransactions within the MDMS transaction is determined before the execution of the transaction.

Before a transaction is executed, it requests all the interfaces at the sites which are involved to arrange the scheduling order of its subtransactions at the local site in order to prevent any inconsistencies its execution may cause.

The MDMS interface only communicates with GTM's of the global transactions that have a subtransaction at its site and can run independently of other GTM's.

Within the framework of this approach, we need to make some assumptions in order to define a boundary within which we can operate. The assumptions about the GTM and the transaction model will be discussed in the following two sections. The correctness criterion which we propose to use will be outlined in section 7.2.4.

7.2.1 Assumptions about the global transaction manager

The following assumptions about the GTM can be made without a loss of generality and without affecting local site autonomy [Ras93b]:

- We assume that the GTM is located at the site at which the global transaction is submitted, and controls the execution of all global transactions submitted at that site. Users access data at remote sites by executing global transactions which make calls to the GTM.
- For each global transaction executed, the GTM will decide which local site or sites should be accessed in order to execute the transaction.
- At each such site, there is a server process (one per site) and the GTM submits the subtransactions to the server if scheduling the subtransaction for execution will not cause database consistency to be violated.
- The server process receives execution requests for subtransactions of global transactions to be executed at site LDB^i, determines their scheduling order, creates interface processes to execute them in the pre-determined scheduling order, and recovers them from errors.
- The local DBMSs do not distinguish between local transactions and global subtransactions executing at its site.
- No assumptions are made about the LDBs or their interfaces so that local autonomy is retained.
- The MDMS schedules subtransactions and not operations, as the basic unit of execution. The local DBMS executes subtransactions as local transactions

- We also assume that a mechanism exists for an interface to exist between the local DBMS and the local server so that operations submitted by the server will be acknowledged by the local DBMS to the server.
- Each local DBMS must also follow concurrency control protocols which ensure *conflict serializability*. Conflict serializability will be referred to henceforth as serializability. Also, local and global transactions, when executed in isolation, preserve database consistency.

7.2.2 Assumptions about the transaction model

Implicit in this architecture and computational model are assumptions about the system architecture and how users interact with the MDMS. The following assumptions have been made [Bar91]:

- *Local autonomy* — The individual DBMSs are assumed to be fully autonomous. They therefore cannot be modified in any way nor can they communicate with each other. [Bar91] also says that autonomy implies that each transaction will execute to termination. In the event of any failure, each DBMS is able to fully recover autonomously and correctly without user input.
- *Heterogeneity* — No assumptions are made about heterogeneity. The user interfaces, data models and transaction management policies of each DBMS may be different. In this dissertation we are concentrating on autonomy and not on heterogeneity.
- *Subtransaction decomposition* — The model assumes that a number of subtransactions execute on various databases on behalf of a global transaction. We will not address the decomposition of global transactions into subtransactions but will assume that some sort of mechanism exists to do these decompositions effectively.
- *Data replication* — Data replication across member databases is not considered in this model.
- *Multiple subtransactions* — A global transaction cannot submit multiple global subtransactions to a single DBMS.
- *Failures* — Media failures that cause part or all of the local database's stable storage to be lost are not considered because the MDMS cannot control the mechanisms employed by the local DBMSs. Therefore, each DBMS must guarantee reliability in the event of such problems.
- *Network* — This model assumes that the underlying network is reliable and that error correction will be handled by the underlying network protocol. It also assumes that network failures like partitioning will be handled by the network layer and that issue was not addressed in this dissertation because it is a research field all of its own.

7.2.3 Multidatabase serializability

We feel that serializability is not appropriate in multidatabase systems as a correctness criteria because it is intended to model transactions contained in a single history and because it limits concurrency unacceptably. A new correctness criteria should accommodate the multiple histories in a multidatabase environment. This correctness criteria should capture both the local histories and the history of global transactions which are not completely contained at a single local database system. The definitions presented in this section define such a correctness criterion.

In Chapter 3 we discussed the concept of conflicting operations and conflicting transactions. We can extend this concept to conflicting global subtransactions:

Definition 7.1 — *Conflicting global subtransaction*

A global subtransaction GST_i^j *directly conflicts* with a distinct global subtransaction GST_k^j if $GST_i^j \rightsquigarrow GST_k^j$ is in any local schedule. A global subtransaction GST_i^j *indirectly conflicts* with another global subtransaction GST_k^j in a local schedule if there exist transactions L_1, L_1,...., L_n in the local schedule such that :

$GST_i^j \rightsquigarrow L_1 \rightsquigarrow .... \rightsquigarrow L_n \rightsquigarrow GST_k^j$.

[Tan93] □

Definition 3.11 defines a serial schedule which is at the core of serializability theory. In order to propose a correctness criterion for multidatabases, the next definition defines the concept of a M-Serial history:

Definition 7.2 — *M-Serial history*

A multidatabase history is M-Serial iff:

1. every $LH \in \mathcal{LH}$ is conflict serializable, and
2. given a $GH = \{GST_1^n, ..., GST_r^m\}$, if $\exists p \in GST_i^k, \exists q \in GST_j^k$ such that $p \prec_{GH} q$, then $\forall k, \forall r \in GST_i^k, \forall s \in GST_j^k, r \prec_{GH} s$.

[Bar90]

□

The first condition states that local histories are conflict serializable. It is not necessary to require that local histories be serial since we assume that each local transaction manager can serialize submitted transactions. The second condition states that if an operation of a global transaction precedes an operation of another global transaction in one local history, then all operations of the first global transaction must precede any operation of the second in all local histories [Bar90].

Definition 7.3 — *Equivalence of histories ($\equiv$)*

Two histories are conflict equivalent if they are defined over the same set of transactions and they order conflicting operations of nonaborted transactions in the same way .

[Bar90] □

The notion of M-Conflicting transactions is also necessary in order to define M-Serializability:

Definition 7.4 — *M-Conflict*

A global transaction GT_j is said to be in multidatabase conflict (M-Conflict) with another global transaction GT_i if any global subtransaction of GT_j conflicts directly or indirectly with any global subtransaction of GT_i in any local schedule at any LDB participating in the multidatabase service. The M-Conflict relation is denoted by: $GT_i \overset{M}{\rightsquigarrow} GT_j$ and the transitive closure is denoted by $GT_i \overset{*}{\rightsquigarrow} GT_j$.

[Tan93] □

Definition 7.5 — *Locally and globally complete histories*

A local history is *locally complete* if all transactions at the LDB have committed or aborted.

A local history is *globally complete* if all transactions executing at the LDB have either committed or aborted and if a subtransaction of a globally committed transaction has aborted, a redo transaction for that subtransaction has committed.

[Meh92c] □

Definition 7.6 — *M-Serializable (MSR)*

If GH_a is a globally complete history at LDB^a, and GST_i is a global subtransaction in GH_a, the GH_a is M-serializable iff for all GH_i, $GT_i \overset{*}{\rightarrow} GT_i$.

A MH is M-Serializable iff it is equivalent to a M-Serial history.

[Tan93] □

The concepts defined above will now be illustrated by means of the following example.

Example 7.1 : Application of the transaction model to the pharmacy example

Lets go back once again to our pharmacy example in Example 4.2. We have two global transactions as follows:

$GT_1 = r_1(d); r_1(e); w_1(s); w_1(d); c_1$

$GT_2 = r_2(d); r_2(u); w_2(s); w_2(d); c_2$

These generate the following subtransactions:

$GST_1^1 : r_1^1(d); r_1^1(e); w_1^1(d); c_1^1$

$GST_1^2 : w_1^2(d); c_1^2$

$GST_2^1 : r_2^1(d); w_2^1(d); c_2^1$

$GST_2^2 : r_2^2(u); w_2^2(s); c_2^2$

We also have local transactions into the DBMSs as follows:

$LT_1^1 : \hat{r}_1^1(e); \hat{w}_1^1(e); \hat{w}_1^1(d); \hat{c}_1^1;$

$LT_1^2 : \hat{r}_1^2(u); \hat{w}_1^2(u); \hat{c}_1^2;$

We will use the $\hat{}$ notation, for example $\hat{r}$, to distinguish local transactions from global subtransactions in this discussion. A possible local history for each LDB will be generated.

$LH^1 = r_1^1(d); r_1^1(e); w_1^1(d); \hat{r}_1^1(e); r_2^1(d); \hat{w}_1^1(e); \hat{w}_1^1(d); \hat{w}_2^1(d); \hat{c}_1^1; c_2^1; c_1^1$

$LH^2 = r_2^2(u); w_2^2(s); w_1^2(s); \hat{r}_1^2(u); \hat{w}_1^2(u); \hat{c}_1^2; c_1^2; c_2^2;$

The following global transaction histories can be derived from these local histories:

$GSH^1 : r_1^1(d); r_1^1(e); w_1^1(d); r_2^1(d); w_2^1(d); c_1^1; c_2^1;$

$GSH^2 : r_2^2(s); w_2^2(s); w_1^2(s); c_1^2; c_2^2;$

The global history is given by $GH = \{ GSH^1 \cup GSH^2 \}$. The multidatabase history is given by $MH = \langle \{LH^1, LH^2\}, GH \rangle$.

We can see that both local histories are serializable. The multidatabase history is not serializable since the global transaction order at each DBMS is inconsistent. If we have a look at the GH tuple:

$$GH = \{r_1^1(d); r_1^1(e); w_1^1(d); r_2^1(d); w_2^1(d); c_1^1; c_2^1;\} \cup \{r_2^2(s); w_2^2(s); w_1^2(s); c_1^2; c_2^2;\}\},$$

we can see that:

$r_1^1(d) \prec_{GH} w_2^1(d)$ at LDB^1 and $w_2^2(s) \prec_{GH} w_2^1(s)$ at LDB^2

which implies that:

$GST_1^1 \prec_{GH} GST_2^1$ and $GST_2^2 \prec_{GH} GST_1^2$

These two serialization orders are contradictory — $DBMS^1$ specifies that global transaction GT_1 precedes GT_2 while $DBMS^2$ specifies the reverse. Thus we can see that although the local histories are serializable, the execution order specified in GH is not, so the MDB history is not M-serializable.

[Bar90] ◇

This example illustrates how difficult it is to ensure correct serialization when GT's and local transactions are present. We will extend this model to recoverability of global histories in Chapter 6.

According to Breitbart *et al* [Bre92d], global serializability requires that schedules at each local site be m-serializable. This has been proved by Bradshaw [Bra93]. The key problem in guaranteeing m-serializability is that local schedules may generate indirect conflicts between global transactions that otherwise would not conflict. Indirect conflicts are hidden from the GTM and may lead to cycles in the global schedule [Tan93].

Various algorithms have been proposed to deal with indirect conflicts. Some of the algorithms assume a failure free environment while others assume that if there were to be a failure, there would be some form of recovery. Some of the algorithms that do not make provision for failure are the forced conflict and ticket schemes proposed by Georgakopoulos *et al* [Geo91b], and the site-locking algorithm proposed by Alonso *et al* [Alo87] and the site-graph testing algorithm by Breitbart *et al* [Bre88] as well as the altruistic locking scheme by Salem *et al* [Sal89].

These algorithms do not guarantee atomicity of distributed global transactions in the presence of failures and will only work if the local sites support some sort of atomic commit protocol. We cannot, however, expect the local site to provide or export its prepare-to-commit operation in order that the GTM may participate in the two phase commit protocol [Tan93].

The algorithm that our model will use will be outlined in the following section.

7.2.4 Global concurrency control

We have reviewed many different concurrency control schemes and many of them suffer from one or more of the following problems:

- they expect specific conditions (e.g. rigorousness of the local database schedules) to be satisfied in the local database systems (e.g. Barker's scheme — Synopsis 4.22),
- they are not failure resilient (e.g. Gligor *et al*, Georgakopoulos and Breitbart *et al*'s schemes — see Synopses 4.1, 4.6, 4.2),
- they violate local database autonomy (e.g. Zhang *et al*'s scheme – Synopsis 4.14),
- they allow local transactions to only update a portion of the available data items (e.g. Breitbart *et al*'s scheme – Synopsis 4.10),
- they require 2PL at the local site (e.g. Wolski *et al*'s scheme — Synopsis 4.4),
- they generate unacceptably high overhead (e.g. Yun *et al*'s scheme - Synopsis 4.13),
- they expect users to specify correct interleavings of subtransaction operations. (e.g. Du *et al*'s scheme – Synopsis 4.18),
- they could result in global deadlock (e.g. Mehrotra *et al*'s RS-correctness scheme – Synopsis 4.21),
- they relax the atomicity requirement of transactions (e.g. Levy *et al*'s scheme – Synopsis 4.31).

We propose to use the customized global concurrency control algorithm as outlined in [Che93]. This algorithm utilizes the semantics of global transactions and the concurrency control strategies of the underlying LDBs to customize a global concurrency control algorithm.

The algorithm combines two-phase-locking and linear ordering of resource locks to permit a deadlock free, totally distributed and correct synchronization of concurrent scheduling order of requests from global transactions.

The execution of the global transaction is performed in two phases.

- In the first phase, the relative scheduling order of a global transaction with respect to other global transactions at each site is determined.
- In the second phase, the global transaction is executed in the relative scheduling order as determined in the first phase.

The *relative scheduling order* (RSO) is determined differently on different LDBs by accommodating and making use of their differences. For example [Che93]:

1. If a global transaction consists of only read-only applications, or MDMS consistency is not required, the RSO is interpreted as *No-Order*.
2. If the underlying LDB supports two-phase commitment, the RSO can be determined by the order of the prepare-to-commit states for subtransactions on the LDB.
3. If local conflicts can be forced in the LDB, the RSO coincides with the order of obtaining the ticket at the local site and requires each global transaction to access the ticket at the local site — thus creating direct conflicts.
4. If there is no value dependency among subtransactions of a global transaction, the RSO can be determined by the commit order of subtransactions on LDBs.
5. If the underlying LDB is rigorous, the RSO can be determined by the commit order of subtransactions at the LDB.
6. If the underlying LDB supports both two-phase-locking and two-phase-commit, and timestamp or commit order is used as the ordering strategy, the RSO can be determined by the prepare-to-commit order.

This algorithm therefore guarantees that the RSO of global transactions on different sites is consistent with their pre-determined relative scheduling order, ensuring that sites have the same RSO at all sites.

It has been proved by Breitbart *et al* [Bre88] that when global transactions have the same RSO at all sites, global serializability is preserved in the presence of local transactions. Bukhres *et al* [Buk93] state that this algorithm preserves quasi-serializability [Du89] and global serializability if underlying global systems are rigorous or if local conflicts can be forced at all local systems.

Barker [Bar90] has proved that quasi-serializability is equivalent to m-serializability and therefore the algorithm satisfies our requirements for MDMS correctness and consistency.

Example 7.2 : Application of the GCC algorithm

Lets go back once again to our pharmacy example in Example 4.2. Using the transaction model introduced in Chapter 3 and extended here, let us refer to the Tonic pharmacy as LDB^1, the Medilots pharmacy as LDB^2 and the Harbour pharmacy

as LDB^3. We will assume that LDB^1 allows users to create relations and update data; LDB^2 is rigorous; and LDB^3 supports two-phase commit.

In order to maintain global serializability, the MDMS server I_1 associated with LDB^3 uses the order of prepare-to-commit states of subtransactions on LDB^3 as the RSO of the subtransactions. MDMS server I_2 associated with LDB^2 uses the commit order of subtransactions on LDB^2 as the RSO of subtransactions, while MDMS server I_1 associated with LDB^1 creates a relation with some data item as the ticket and uses the order of update operations on the ticket as the RSO of subtransactions.

If all global transactions are read-only, the MDMS administrator can change the strategy adopted by the servers to No-Order. In that case, all servers will allow subtransactions to execute in random order.

[Che93] ◇

The customization of the GCC strategy is determined by the semantics of transactions as well as the transaction management strategy employed by the local database system [Che93].

7.2.5 Reliability in a multidatabase environment

As discussed in Chapter 5, the crux of the matter with respect to reliability is that an effective global commit protocol be used. If the local database systems all provide a prepare-to-commit state, this is trivial but otherwise, a global commit protocol has to be improvised. The commit protocols discussed in Chapter 5 employ one of the following devices:

- they violate local autonomy (eg. Barker & Özsu, Georgakopoulos, Kim *et al*, Muth & Rakow, Perrizo *et al* and Soparkar *et al* [Bar91, Geo91a, Kim93, Mut91, Per91, Sop91b]), or
- they limit the types of transactions allowed, or the data accessed (eg. Breitbart *et al*, Levy *et al*, Du *et al* and Wolski *et al* [Bre92b, Lev91a, Vei92, Wol90]), or
- they assume that a subtransaction abort due to commit operation failure cannot occur (eg. Barker & Özsu, Georgakopoulos and Kim *et al* [Bar91, Geo91a, Kim93]), or
- they assume that if an MDMS site fails during recovery of a local database system then the LDBMS blocks until the MDMS site is recovered (eg. Barker & Özsu and Kim *et al* [Bar91, Kim93]), or
- they use a new transaction/correctness model. Some models weaken transaction atomicity (eg. Mehrotra *et al* [Meh92b]).

In Chen *et al*'s model [Che93], the server provides the necessary synchronization but this means that Chen *et al* assume that transactions can be split up into separate steps[1], which may not always be possible. Any global commit protocol which does not allow the local database to commit at will, essentially violates local autonomy and Chen *et al*'s model does this to a lesser degree than other approaches discussed in Chapter 5.

We recommend the global commitment method outlined by Chen *et al* [Che93] as it was developed specifically for the architectural model we decided upon in the previous section.

7.2.6 Recovery in a multidatabase environment

In the recovery procedures cited in Chapter 6, we can note the following:

- Some assume something about the local database schedules (eg. Georgakopoulos *et al*, Kang & Keefe and Hwang *et al* assume that the local databases use 2PL or strict 2PL [Tun92, Kan93, Hwa94]).
- Some restrict access to data by local and global transactions (eg. Breitbart *et al* and Hwang *et al* [Bre92d, Hwa94]).
- Some expect to have exclusive access to the local database after a site failure (eg. Barker *et al* and Georgakopoulos [Bar90, Geo90]).
- Some violate the autonomy of the local database systems (eg. Pu and Yoo & Kim [Pu88, Yoo95]).
- Some need compensating transactions (eg. Garcia-Molina *et al* [Gar87]).
- Some do not make provision for site failures (eg. Chen *et al* [Che93]).

Chen *et al*'s recovery procedure will work for the transaction model proposed in the previous chapter. It has been implemented in the InterBase system at Purdue University. The architecture of the system has been outlined in section 7.2.

The reason that this crash recovery method has been decided upon is because it does not violate the autonomy of the local database systems and it does not require any type of schedule or locking protocol from the underlying database system. The exact recovery protocol has been described in section 6.6.5.

7.2.6.1 Failure and how Chen *et al*'s recovery protocol succeeds

Transaction failures : Transactions will not fail globally because the serialization order is determined in advance and therefore no problem can occur. Global transactions will not fail locally because deadlock cannot occur with the particular concurrency control algorithm we use and conflicts also cannot occur.

[1] execution step, confirm step and undo step

Failures at local sites : There are three components at the local site: the interface, one or more servers and the local DBMS. We have outlined the crash recovery protocol but we need to have a look at whether MDB consistency is maintained in the face of failures.

If the interface fails and we say for argument sake that there are three servers presently active for three subtransactions of a global transaction which is supported by a GTM at another site. If the interface fails and the GTM manages to reactivate it, it will restore itself to its precrash state and the global consistency will not be compromised. If the GTM fails to reactivate the interface, it will abort the subtransactions and itself and once again the global consistency will be maintained. So whichever way it happens, either all subtransactions will be aborted or all will be committed.

If a server fails and the crash-recovery protocol is followed, then once again either all subtransactions will commit or all will abort.

MDMS site failures : There is no *one* MDMS site in Chen's model as the GTM is distributed. However, if one of the GTMs fail, it will be detected by the interface and if the interface fails to reactivate it, it will abort all subtransactions at its local site and also notify the other interfaces at the other sites so all subtransactions can be aborted. Once again, consistency is maintained. If some of the subtransactions have already committed, compensate transactions can be submitted to undo them.

7.2.6.2 Comment

Having evaluated these research efforts, one comes to the conclusion that any recovery scheme will *have* to violate autonomy in order to be effective [Hwa94]. Chen *et al*'s scheme does not seem to violate autonomy but also does not address site failures. If Chen *et al*'s scheme were to be extended to handle site failures I have no doubt that they would also require some sort of exclusive period in order to recover global transactions without interference from local transactions. Yoo *et al*'s scheme does not require an exclusive access period because he expects *all* transactions to be routed via the local stub which is also a violation of local autonomy. The recovery aspect of multidatabase transaction management still needs a great deal of work as the solutions proposed in the literature are still fairly unsophisticated.

7.3 Summary

In this chapter we evaluated the transaction management schemes in the core group and outlined the relative strengths and weaknesses of each scheme. We then proposed a transaction processing model for multidatabase systems which scores well in the autonomy violation stakes, is reliable, and satisfies the m-serializability correctness criterion. The various aspects of transaction management which we studied were briefly discussed and the various

shortcomings and strengths of the work done by different researchers were also outlined. The proposed transaction processing model was chosen on the basis of the optimal properties of transaction management identified during the course of this research. The following chapter will summarize the work done in, and the conclusions reached as a result of, this research.

Chapter 8

Conclusion

8.1 Method of research

I set out to study transaction management in multidatabase systems. In order to define the scope of my research I first attempted to understand the parameters which distinguish a multidatabase from a distributed database system. I found that there were basically three classification methods, one which classified multidatabases according to architectural differences [Bel92], another which classified them according to degree of autonomy, heterogeneity and distribution [Özs90] and yet another which classified them according to how tightly the participating local database were coupled [Bri92]. I decided to integrate the methods in order to arrive at a single classification method which incorporates all these aspects of multidatabase implementations which is presented in Chapter 2. In the same chapter the autonomy dimension was also identified as the one which had the greatest relevance when considering various multidatabase transaction management algorithms. A quantification method was introduced in order to measure this important dimension.

I then studied the efforts of several researchers into transaction management in multidatabase systems and found that there were many totally divergent approaches each of which had different weaknesses and strengths. This led to the decision to limit my research to a core group of eight different research groups and to study these schemes in detail. I have given an overview of the essential features of each of these schemes in Chapter 4. The concurrency control mechanisms for multidatabases is a widely researched field and I decided to give a fairly comprehensive overview of the work done in that area. Although the field of global deadlock in multidatabases lies on the fringe of the transaction management research area, I decided not to discuss it in great detail but have given a summary of the latest research for the sake of completeness.

Not as much research has been done thus far into the fields of reliability and recoverability in multidatabase systems. In order to achieve reliability, one has to guarantee transaction atomicity and have an efficient crash recovery protocol. Transaction atomicity is achieved by implementation of a global commit protocol. This is not a trivial task in

Researcher	Reference	Method
Bell & Grimson	[Bel92]	Architectural differences
Özsu & Barker	[Özs90]	Degree of autonomy heterogeneity & distribution
Bright *et al*	[Bri92]	How tightly the local databases are coupled

Table 8.1: Classification Schemes Studied

multidatabase systems. Chapter 5 addresses the transaction atomicity aspect and examines research into global commit protocols. The crash recovery aspect is discussed in Chapter 6 where recent research into crash recovery protocols is summarized.

Recoverability goes hand in hand with reliability because if recoverability requirements are satisfied then the correctness of the multidatabase can be guaranteed in the case of failure. Chapter 6 takes a look at the types of failure which can be expected in a multidatabase and then extends the transaction model which was introduced in Chapter 3 to incorporate recoverability aspects.

8.2 Issues Studied and Achievements

In tables 8.1 & 8.2, the work in this thesis is summarized.

The classification schemes shown in table 8.1 were studied and were merged to form a single taxonomy of multidatabase systems which was presented in Chapter 2.

The members of the transaction management core group shown in table 8.2 were chosen as the subject of this research because they serve as a good representative sample of present research in the field.

The table shows that there is a definite move towards utilizing the client server approach in multidatabase systems. Most of the latest research seems to be moving in that direction. It is also interesting to note that none of the schemes manage to maintain full autonomy but the scheme by Chen *et al* violates autonomy to the least extent. We can also note that most of the schemes use serializability as a correctness criterion in spite of the rigidity of that approach but there is a move towards m-serializability and the equivalent quasi-serializability in the latest research.

Each scheme in the core group has a different approach with respect to global concurrency control utilized with a widely divergent group of schemes being used. The global commit protocols chosen, on the other hand mostly seem to lean towards the two-phase commit protocol. The scheme either assumes that the local database provides support for it, or they use their local server to emulate a two-phase commit protocol. A notable exception here is the scheme by Chen *et al* which uses a semantic based commit protocol. The

Researcher	Reference	Description	Autonomy Violation	Correctness Criterion	Deadlock Handling	Global Concurrency Control	Global Commit Protocol	Crash Recovery
Barker & Ozsu	[Barker 1990, Ozsu 1991]	Basic MDB model	1.0 Semi-autonomous	M-Serializability	Prevention	Serializability graphs	Emulate 2PC	Retry & exclusive access period
Pu	[Pu 1988]	Hierarchy of superdata-bases	1.25 Non-autonomous	Serializability	Not addressed	Violation of local autonomy	2PC	Redo
Breitbart *et al*	[Breitbart 1988, Breitbart 1986, Breitbart 1985, Breitbart 1987]	Replicated data model		Serializability	Global site graph	Site Graph, Rigorous schedules	2PC	
	[Breitbart 1995]	Server model	1.0897 Semi-autonomous	Serializability	Global wait-for graph	Optimistic ticket method	2PC between GTM and server	Done by local server
Elmagarmid *et al*	[Elmagar 1988, Elmagar 1987, Elmagar 1986, Du 1989, Elmagar 1990a]	Stub approach	1.116 / 1.73 Semi-autonomous	Quasi-serializability	Deadlock free	Serialization events	Not applicable	Not applicable
Chen *et al*	[Chen 1993]	Distributed MDMS	0.829 Semi-autonomous	Quasi-serializability	Deadlock free	Linear ordering of resources	Semantic Based Commit	Done by local interface - compensate
Kang & Keefe	[Kang 1993]	Distributed GTMs	0.93 Semi-autonomous	Serializability	Deadlock free	Distributed strict timestamp ordering	Emulated 2PC	Redo
Garcia-Molina *et al*	[Garcia 1987]	Sagas	0.93 Semi-autonomous	Not applicable	Not addressed	No scheme needed	Not applicable	Compensate or redo
Yoo & Kim	[Yoo 1995]	Client server model	1.435 Semi-autonomous	Serializability	Detect & Resolve	Not addressed	Reliable 2PC	Handled by local agent

Table 8.2: Transaction Management Schemes Studied

crash recovery protocols used by the core group mostly violate local autonomy, especially after restart of a failed site. Some also use the compensation method to undo the effects of committed transactions and this is less than ideal because local transactions may see data values they are not meant to see.

In Chapter 7 of this dissertation, each of the core groups' transaction management schemes was evaluated according to the autonomy quantification method. The scheme which was identified as the best possible scheme from the autonomy point of view was the scheme presented by Chen *et al* [Che93]. Barker & Özsu's m-serializability was chosen as the notion of correctness for our multidatabase system and the transaction processing model was extended by incorporating the work of Tang [Tan93], Mehrotra *et al* [Meh92c] and Barker [Bar90] in order to formalize the correctness criterion for our chosen multidatabase system.

8.3 Future Research

During the course of my research I gained a good understanding of multidatabase systems, various architectures, transaction management schemes and above all the unique problems faced by the multidatabase system designer.

Several potential future areas for research exist in this relatively new field. The areas of global concurrency control, multidatabase architectures, and global commitment have been fairly well researched but the areas of fault tolerance and safety of multidatabase systems have not been researched to that extent. Fault tolerance improves reliability of a system by replicating service providers so that the system can continue to function and produce correct results even if some components fail. The safety aspect would have to examine the safety of each local database which would now be accessible to many more users which the database system does not have the power to authorize. Mechanisms should be put into place which provide access control to the multidatabase system as a whole and to control access to possibly sensitive data by global users. The probability of security violations increases when a database system joins a multidatabase structure and these issues need to be addressed in the light of the current increase in computer crime.

In the light of technological advances which will make multidatabases more and more common in organizations, the need for further research is indubitable.

Appendix A

Glossary

2LSR — Two-Level-Serializable : A schedule is 2LSR if each DBMS generates serializable schedules and the restriction of the schedule to only global transactions is serializable.

2PC — Two Phase Commit : A global commitment protocol where the commit decision is first sent to all participants and then after they have all replied, another message is sent and the transactions are submitted.

2PL — Two Phase Locking : A locking protocol where all locks are obtained before operations are carried out and relinquished after all work has been done.

ACA — Avoids Cascading Aborts : A requirement for a schedule which requires transactions to only read items written by committed transactions.

ACP — Atomic Commitment Protocol : A protocol which ensures that all subtransactions of a global transaction either commit or abort.

AGCA — Avoids Global Cascading Aborts : A requirement which is equivalent to the ACA but which applies to global multidatabase schedules.

ALCA — Avoids Local Cascading Aborts : A requirement which is equivalent to the ACA but which applies to local schedules in a multidatabase environment.

CAD/CAM — Computer Aided Design / Computer Aided Manufacturing : Software to support computer aided design and manufacturing.

CO — Commitment Ordering : Property of a local schedule which ensures global serializability in a multidatabase environment.

CPU — Central Processing Unit

DBs — Databases : A collection of related data.

DBA — Data Base Administrator : The person in control of the database.

DBMSs — Database Management Systems : A collection of programs that enables users to create and maintain a database.

DDB — Distributed Database — Database systems that are interconnected by a communications network.

DDBMS — Distributed Database Management System : Software used to implement a distributed database system.

DSTO — Distributed Strict Timestamp Ordering : A global concurrency control scheme proposed by Kang & Keefe.

ESR — Epsilon-Serializability : An alternative to global serializability in a multidatabase system.

FC2PL — Forced Conflict two Phase Locking : A combination of 2PL and forced local conflicts in order to produce a schedule which is globally serializable.

FT — Flexible Transaction : A transaction which is provided along with various specifications about transaction states and transitions between those states.

GDD — Global Deadlock Detector : A program for detecting the presence of a global deadlock in a multidatabase environment.

GlST — Globally Strict : A requirement which is equivalent to the strict requirement but which applies to global multidatabase schedules.

GRC — Global Recoverable : A requirement which is equivalent to the recoverable requirement but which applies to global multidatabase schedules.

GST — Global Subtransaction : The part of a global transaction which is sent to a single database system for execution.

GTM — Global Transaction Manager : The software which controls global transactions in a multidatabase system.

IR — Isolation of Recoveries : A correctness criterion for schedules. A schedule is IR if no transaction sees both the compensated for effects, as well as the committed effects of other transactions.

ITM — Implicit Ticket Method : A refinement of OTM that eliminates ticket conflicts.

LDB — Local Database system : A database system which is a member of a multidatabase.

LDBS — Local Database Management System : The software which is in place in a local database system to create and maintain the database.

LDM — Local Data Manager (also known as LRM) : The data manager in the local database system.

LTM — Local Transaction Manager : The transaction manager at the local database system.

LS — Local Scheduler : The scheduler at the local database system.

LST — Locally Strict : A requirement which is equivalent to the strictness requirement but which applies to local schedules in a multidatabase environment.

LRC — Local Recoverable : A requirement which is equivalent to the recoverable requirement but which applies to local schedules in a multidatabase environment.

LRM — Local Recovery Manager : The recovery manager at the local database system.

MDB — Multidatabase : A database system made up of pre-existing, geographically distributed, heterogeneous, semi-autonomous database systems.

MDBS — Multiple Database System : A system made up of multiple pre-existing, homogenous or heterogeneous, distributed or centralized database systems.

MDMS — Multidatabase Management System : The software which controls access to data in a multidatabase system.

MDS — Multidatabase System : The multidatabase and the multidatabase management system.

MSR — M-Serializable : A correctness criteria for multidatabase systems which is weaker than global serializability.

OTM — Optimistic Ticket Method : OTM is a multidatabase transaction management mechanism that guarantees global serializability by permitting execution of multidatabase transactions only when their relative serialization order is the same in all participating LDBs.

PTM — Pessimistic Timestamp Method : A concurrency control algorithm proposed by Yun and Hwang.

PWSR — Predicate-Wise Serializability : An alternative to global serializability.

QSR — Quasi-serializable : An alternative to global serializability.

RC — Recoverability : A requirement for a schedule which states that no transaction may commit unless the transactions which wrote data items which this transaction read have committed.

RDA — Remote Data Access : To access data at a remote site.

RSO — Relative Scheduling Order : A method introduced by Chen which combines two-phase locking and the linear ordering of resources.

SG — Serialization Graph : A method of ensuring conflict serializablity of schedules.

SGT — Serialization Graph Testing : Methods whereby the serialization graph is tested.

SQL — Structured Query Language : A query language for extracting query results from databases.

ST — Strictness : A requirement for a schedule which states that no transaction may read or write a data item unless the transactions which wrote those data items have committed.

TBSG — Transaction Blocked at Site Graph : A global deadlock detection scheme for multidatabase systems.

TO — Timestamp Ordering : A concurrency control scheme which assigns a timestamp to data items and then compares the timestamps when conflicts occur.

Appendix B

Terms used in Formal Transaction Modelling

a_i	abort of transaction i.
$\mathcal{BS}_i$	Base set of transaction i.
$b(T_i)$	The time at which T_i became blocked.
c_i	commit of transaction i.
$C(S_T^c)$	Complete schedule.
GH	Global history of a multidatabase.
GSH^k	Global subtransaction history at site k of a multidatabase.
GST_i^j	Global subtransaction j of global transaction i.
GT_i	Global transaction i.
$\mathcal{GT}_i$	The set of all global transactions in the multidatabase.
$\mathcal{LDB}^i$	The set of all the data items at site i.
LH^k	Local history of a site k in a multidatabase.
$\mathcal{LH}$	The set of all local histories in a multidatabase.
LT_i^j	Local transaction i at site j of the multidatabase.
$\mathcal{LT}^j$	The set of all local transactions at a local database system j.
$\mathcal{LT}$	The set of all local transactions in the multidatabase.
$\mathcal{MDB}$	The set of all data in the multidatabase.
MH	Multidatabase history.
N_i	Termination condition for T_i.
$O_{ij}(X)$	Operation O_j of transaction T_i.
OS_i	Set of all the transactions in T_i.
r	Read-item.
$rl_i(X)$	The time at which data item X was last locked for reading.
$\mathcal{RS}_i$	Read set of transaction i.
$rts(X)$	Read timestamp of data item X.
S	Schedule; ordering of operations of transactions T_1,...,T_n.

S_T^c	Complete schedule.
$ts(T_i)$	Timestamp of transaction T_i.
T_i	Transaction i.
w	Write-item.
$wl_i(X)$	The time at which data item X was last locked for writing.
$\mathcal{WS}_i$	Write set of transaction i.
$wts(X)$	Write timestamp of item X.
Σ_i	Domain of T_i.
$\prec_i$	Binary relation indicating the execution order of operations in a transaction T_i.
$\equiv$	Equivalence.
$\models$	Leads to
$\rightsquigarrow$	Conflicts.
$\overset{*}{\rightsquigarrow}$	Transitive Closure of Conflicts
$\overset{d}{\rightsquigarrow}$	Direct Conflict
$\overset{i}{\rightsquigarrow}$	Indirect Conflict
$\overset{M}{\rightsquigarrow}$	M-Conflicts
$\overset{\diamond}{\rightsquigarrow}$	Transitive Closure of M-Conflicts

Appendix C

Commit Protocols

C.1 Two-Phase Commit

One very well known atomic commitment protocol is the two-phase commit protocol. This protocol is initiated by a coordinator - in our case the MDMS. The coordinator sends a message to all participants telling them to prepare to commit. Each participant replies with a vote indicating whether or not it is ready to commit. Once a participant replies that it is ready to commit, the decision cannot be reversed (i.e. the local concurrency control at the participant site cannot abort the participant). If the coordinator receives messages from all participants saying that they are ready to commit, it decides to commit and sends a second message to each participant telling it to commit [Ber93].

C.1.1 Two Phase Commit Protocol

1. *Phase One*:

   (a) The coordinator send a *prepare* message to all participants.

   (b) Each participant waits until it receives the *prepare* message from the coordinator. It then votes *ready* or *aborting* and sends the corresponding message to the coordinator, depending on whether it is a pessimistic or optimistic control:

   - *Pessimistic control* — If the participant has been aborted, it decides to abort and sends an *aborting* message. If not, it sends a *ready* message and enters a state in which it cannot be aborted by the local control.
   - *Optimistic control* — The participant executes a validation phase. If validation fails, it decides to abort and sends an *aborting* message; otherwise it sends a *ready* message and enters a state in which it cannot be aborted by the local control

2. *Phase Two*:

   (a) If the coordinator receives at least one *aborting* vote, it decides to send an abort message to all participants. If all votes are ready, it decides to commit and sends a *commit* message to each participant. Then it terminates.

   (b) The actions taken by a participant when it receives an *abort* or *commit* message depend on whether it is an immediate-update or deferred-update control:

   - *Immediate-update control* — If a participant receives an *abort* message, it decides to abort and rolls back any changes it made to the database. If it receives a *commit* message, it commits. In both cases it releases all locks, discards its write-ahead log and terminates.
   - *Deferred-update control* — If a participant receives an *abort* message, it aborts, discards its intentions list and terminates. If it receives a *commit* message, it commits, executes its write phase, discards its intentions list and terminates. If control is pessimistic, locks must also be released [Ber93].

C.1.2 Properties of an atomic commit protocol

Failure can occur before the two-phase commit protocol is initiated or while it is being performed. To cope with failure we need the atomic commit protocol to have the following properties [Ber93]:

1. All sites that reach a decision must reach the same decision.
2. If there are no failures and all sites vote to commit, the decision of all sites will be to commit.
3. If any site votes to abort, no site can decide to commit (even if failures occur)
4. Once a site has made a decision to commit or abort, it cannot reverse that decision (even if failures occur).

It is desirable for an atomic commit protocol to be robust, which means that:

> For all executions in which failures of the given type have occurred, if all failures are repaired and no new failures occur, all sites will eventually reach a consistent decision.

A site must depart from the normal execution of a protocol when a failure occurs. Protocols must try to recover from failures. A timeout protocol is executed if a site times out while waiting for a message. A restart protocol is executed if a site is recovering from a crash. The site therefore needs to maintain a log of significant events that occur during the commit protocol. This log makes the recovery process possible.

C.1.3 Problems with two-phase commit

The two phase commit protocol exhibits blocking under certain circumstances. [Ber93] shows that any atomic commit protocol that is robust for partition failures exhibits blocking. It can also be shown that protocols which return undeliverable messages to the sender which can deal with networks being partitioned into two partitions cannot deal with general partitioning [Ber93].

When false timeouts occur, an even stronger negative result about blocking can be deduced. From properties 2 and 3 of an atomic commit protocol as outlined in the previous discussion it follows that, to reach a commit decision, the vote of every operational site must be considered. If a particular site is slow to send its vote, other sites waiting for the decision may not abort (property 2). Unfortunately, a slow response cannot be distinguished from failure. Since waiting for a failed site is blocking, we can conclude that blocking can occur when even one failure can occur [Ber93].

C.1.4 Timeout protocol for two-phase commit

During the execution of two-phase commit, in that period after a participant site has sent its vote to commit but before it has enough information to know what the decision of the MDMS site is, it is said to be uncertain; that period is called the uncertainty period.

To extend the two-phase commit protocol to deal with failures, we must supply a timeout protocol for each waiting period and a restart protocol to cope with site failure.

If a participant times out while waiting at step (b) of Phase 1, it can be certain that no decision to commit has yet been taken at any site (since it has not yet voted). Hence it can decide to abort and thus prevent any site from reaching a commit decision since such a decision requires ready votes from all sites.

A similar situation exists if the coordinator times out while waiting at step (a) of phase 2, and hence the coordinator can decide to abort and send an abort message to all participants.

If a participant times out at step (b) of phase 2, the situation is far worse, since it is in its uncertain period. If this occurs, either the coordinator has crashed or a network partition has occurred and the participant has been separated from the coordinator. The coordinator may have decided to commit but before it could communicate that to the participants, a network condition occurred. The participant must block until it receives a message from the coordinator communicating the decision of the coordinator.

We can summarize the timeout protocol as follows:

- Timeout at step (b) of phase 1: The participant decides to abort.
- Timeout at step (a) of phase 2: The coordinator decides to abort and sends an abort message to the participants from whom it received a ready vote.
- Timeout at step (b) of phase 2: The participant must block because it is not permitted to communicate with other participants.

C.2 Three-Phase Commit Protocol

The three-phase commit protocol is an extension of the two-phase commit protocol that does not block when sites fail. If a network becomes partitioned, this protocol is still not robust enough to handle it.

C.2.1 Three-phase commit with no failures

1. The coordinator sends a *prepare* message to all participants.
2. Each participant waits until it receives the *prepare* message from the coordinator. It then votes *ready* or *aborting* and send the corresponding message to the coordinator. If the vote is aborting, it decides aborts.
3. The coordinator waits until it receives votes from all participants. If at least one vote is *aborting*, it sends an *abort* message to all participants. If all votes are *ready*, it sends a *precommit* message to each participant.
4. Each participant that voted *ready* waits to receive a message from the coordinator. If that message is *precommit*, it sends an *acknowledge* message to the coordinator. If the message is *abort*, it aborts.
5. If the coordinator sent a *precommit* message in step 3, it waits until it receives an *acknowledge* message from each participant. Then it decides to commit and sends a *commit* message to each participant.
6. Each participant that sent an *acknowledge* message waits until it receives a *commit* message and then commits.

The uncertain period of a participant starts when it sends a ready message at step 2 and ends when it receives the precommit or abort message at step 4. Thus, while a participant is waiting at step 4, it is uncertain. The uncertain period of the coordinator starts when it sends the first precommit message and ends when it receives the first acknowledge message. While a participant is waiting at step 6 for the commit message, we say it is committable.

The protocol has the property of not being blockable and an operational site will always proceed to completion. We need to specify a timeout protocol to deal with failures.

C.2.2 Timeout protocol for three-phase commit protocol

- Timeout at step 2: The participant decides to abort.
- Timeout at step 3: The coordinator decides to abort and sends an *abort* message to every participant from which it received a *ready* vote.
- Timeout at step 4: The participant executes the termination protocol.

- Timeout at step 5: The coordinator decides to commit and sends a *commit* message to every participant from which it received an acknowledge message.
- Timeout at step 6: The participant executes the termination protocol.

The termination protocol is executed only when the coordinator has failed. The basic idea is that the operational participants elect one of themselves to be the new coordinator.

The election process is described by Bernstein *et al* [Ber93]. Once a new coordinator is elected, it polls all participants to determine where they were in the commit protocol. The new coordinator will then carry out the termination protocol in order to resolve the situation.

C.2.3 Termination protocol for three-phase commit

1. The operational sites elect a new coordinator.
2. The new coordinator polls the operational sites to find out where they are in the commit period.
3. The new coordinator takes the following actions:
   - If any participant has not voted or has aborted, the new coordinator decides to abort and sends an *abort* message to all participants.
   - If any participant has committed, the new coordinator decides to commit and sends a *commit* message to all participants.
   - If all operational participants are uncertain, an *abort* message is sent to all participants.
   - If some participant is commitable but none have committed, the new coordinator sends *pre-commit* messages to all uncertain participants, waits for them to send *acknowledge* messages and then sends commit messages to all participants.

The robustness of the three-phase commit protocol is based on an assumption of no partitions in the network. A partition may result in one partition being uncertain and the other having committed participants. The sites in each partition could decide on different courses of action. Breitbart *et al* have designed a protocol which is robust for partitioning but loses its non-blocking property.

C.3 Multidatabase Two-Phase Commit

Figure C.1 depicts the state diagram of the multidatabase two-phase commit protocol.

The MDMS prepares global transactions for submission to the local database systems (*initial* state). Once all GST's have been submitted, the GT is moved to the *wait* state. If

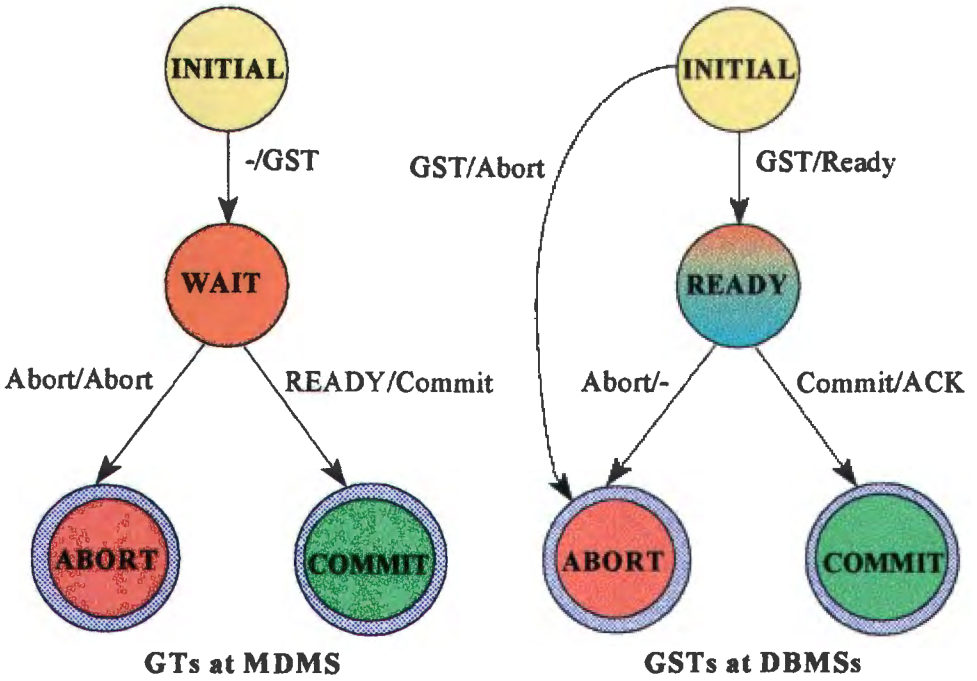


Figure C.1: State Diagram for multidatabase two-phase commit

[Bar90, p103]

all the GSTs become ready the GT is moved to the *commit* state. If any one of the GSTs does not become ready the GT moves to the *abort* state [Bar90].

Once a GST is submitted to a LDB it is in the *initial* state. The GST remains in this state until it decides to abort upon which it moves to the *abort* state, or it is ready to commit, in which case it moves to the *ready* state. Once the GST is in the *ready* state it waits for the final commit decision from the MDMS. If the GT commits, a message is sent to the local level so the GST can move to the *commit* state. The local level acknowledges the *commit* instruction by means of an *ack* message. If the GT is aborted, the GST moves to the *abort* state [Bar90].

The difficulty lies in maintaining the state transitions at the local database systems. The problem is that a lot of local DBMSs may not support 2PC.

C.4 Byzantine Generals Problem

The Byzantine Generals problem considers the situation where one has various distributed processors and faulty processors are actively 'traitorous' and can send any message to another process.

A set of units of the byzantine army is preparing for action against an enemy. Each unit is commanded by a general and these generals communicate with each other by sending messages over telephone lines. The messages are assumed to reach the other end uncorrupted and it is assumed that lines do not fail.

The generals must agree on a course of action. The algorithm for reaching a byzantine agreement between distributed processors assumes that one of the processors is the commander and the others are lieutenants. The algorithm is as follows:

1. The commander sends his decision.
2. A lieutenant relays the commander's decision to every other lieutenant.
3. Upon receiving both the direct message from the commander and the relayed messages from the lieutenants, the lieutenant decides by majority voting on the messages.

This algorithm can be applied to the global commitment problem in the multidatabase situation where the MDMS is the commander and the member database systems are the lieutenants. The algorithm aims to reach agreement in any network in spite of malicious failures.

Appendix D

ANSI-SPARC Architecture

The majority of commercial databases today are based on the ANSI-SPARC architecture which divides a system into three levels, *internal*, *conceptual* and *external*, as illustrated in Figure D.1. The goal of this architecture is to separate the user applications and the physical database.

- The *conceptual level* has a conceptual schema and represents the community view of the data in the database, referred to as a global logical view. The conceptual schema hides the details of physical storage structures and concentrates on describing entities, data types, relationships, user operations and constraints.
- The users view the data through the external schema defined at the *external level.* The external level includes a number of external schemas or user views. Each external schema describes the part of the database that the particular user or group of users is interested in and hides the rest of the database from that user group.
- The *internal level* has an internal schema and is a low-level description of the data in the database and provides an interface with the operating system's file system, which is ultimately responsible for accessing database files. The internal level is concerned with specifying what data items are to be indexed, what file organization technique to use, how the data is to be clustered, and so on.

Most DBMSs do not separate the three levels completely, but most of them support the three-schema architecture to some extent. The three schemas are only descriptions of data; the only data that actually exists is at the physical level. The DBMS must transform a request specified on an external schema into a request against the conceptual schema, and then into a request against the internal schema for processing over the stored database. The process of transforming requests between levels is called mapping [Meh92d].

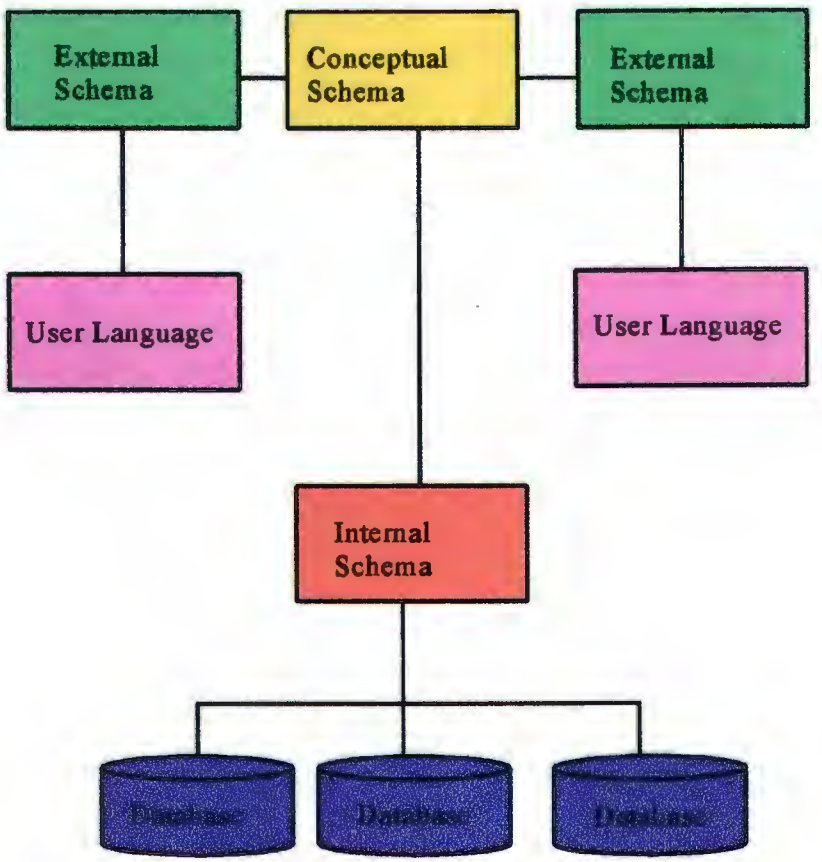


Figure D.1: The ANSI-SPARC three-level architecture

[Bel92, p.17]

References

[Alo87] ALONSO R, GARCIA-MOLINA H & SALEM K. 1987. Concurrency control and recovery for global procedures in federated database systems. *Data Engineering.* 10(3):5-11.

[Alo94] ALONSO G, VINGRALEK R, AGRAWAL D, BREITBART Y, ABBADI A E, SCHEK H & WEIKUM G. 1994. Unifying Concurrency Control and Recovery of Transactions. *Information Systems.* 19(1):101-115.

[Bad92] BADRINATH B R & RAMAMRITHAN K. 1992. Semantics-based concurrency control: Beyond commutativity. *ACM Transactions on Database Systems.* 17(1):163-199.

[Bar90] BARKER K. 1990. Transaction Management on Multidatabase Systems. Doctor of Philisophy Thesis. University of Alberta.

[Bar91] BARKER K & ÖZSU M T. 1991. Reliable Transaction Execution in Multidatabase Systems. In: *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems,* edited by Y Kambayashi, M Rusinkiewicz & A Sheth. Los Alamitos: IEEE Computer Society Press. p344-347.

[Bar94] BARKER K. 1994. Quantification of autonomy on Multidatabase systems. *Journal of Systems Integration.* 4(2):151-169.

[Bat92] BATRA R K, RUSINKIEWICZ M & GEORGAKOPOULOS D. 1992. A Decentralized Deadlock-free Concurrency Control Method for Multidatabase Transactions. In: *Proceedings of the 12th International Conference on Distributed Computing Systems.* Los Alamitos, IEEE Computer Society Press. p72-79.

[Bel92] BELL D & GRIMSON J. 1992. *Distributed Database Systems.* Great Britain. Addison Wesley.

[Ber81] BERNSTEIN P A & GOODMAN N. 1981. Concurrency Control in Distributed Database Systems. *Computing Surveys.* 13(2):185-221.

[Ber87] BERNSTEIN P A, HADZILACOS V & GOODMAN N. 1987. *Concurrency Control and Recovery in Database Systems.* Addison Wesley Publ Company. Reading, Massachusetts.

[Ber93] BERNSTEIN A J & LEWIS P M. 1993. *Concurrency in Programming and Database Systems.* London. Jones and Bartlett Publishers, Inc.

[Ber90] BERSON J S. 1990. Generic RDA Editor. *Information Processing Systems - Open Systems Interconnection — Remote Database Access - Part 1: Generic Model, Service and Protocol (ISO/IEC JTC 1/SC 21 WG3).* ISO 1990.

[Bra93] BRADSHAW D P, LARSON P A & SLONIM J. 1993. Concurrency Control in Multidatabase Management Systems. *Research Report CS-93-34,* Department of Computer Science, University of Waterloo.

[Bre85] BREITBART Y & PAOLINI P. 1985. Session chairman overview — the multibase session. In: *Distributed Data Sharing Systems,* edited by F A Schreiber & W Litwin. North Holland.

[Bre86] BREITBART Y, OLSON P & THOMPSON G. 1986. Database integration in a distributed heterogeneous database system. In: *Proceedings of the 2nd International Conference on Data Engineering.* Los Angeles, CA.

[Bre87] BREITBART Y, SILBERSCHATZ A & THOMPSON G. 1987. An update mechanism for multidatabase systems. *Q. Bull IEEE TC on Data Engineering* 10(3):12-18.

[Bre88] BREITBART Y & SILBERSCHATZ A. 1988. Multidatabase Update Issues. *SIGMOD Record.* 17(3):135-142.

[Bre90a] BREITBART Y, SILBERSCHATZ A & THOMPSON G R. 1990. Reliable Transaction Management in a Multidatabase System. *SIGMOD Record.* 19(2):215-224.

[Bre90b] BREITBART Y. 1990. Multidatabase Interoperability. *ACM SIGMOD Record.* 19(3): 53-60.

[Bre91a] BREITBART Y, GEORGAKOPOULOS D, RUSINKIEWICZ M & SILBERSCHATZ A. 1991. On Rigorous Transaction Scheduling. *IEEE Transactions on Software Engineering.* 17(9):954-960.

[Bre91b] BREITBART Y, LITWIN W & SILBERSCHATZ A. 1991. Deadlock Problems in a Multidatabase Environment. In: *Proceedings of COMPCON Spring '91 Digest of Papers.* Los Alamitos: IEEE Computer Society Press. p145-151.

[Bre92a] BREITBART Y & SILBERSCHATZ A. 1992. Strong Recoverability in Multidatabase Systems. In: *Proceedings of the Second International Workshop on Research Issues on Data Engineering: Transaction and Query Processing.* Los Alamitos: IEEE Computer Society Press. p170-175.

[Bre92b] BREITBART Y, SILBERSCHATZ A & THOMPSON, G R. 1992. Transaction Management Issues in a Failure-Prone Multidatabase System Environment. *VLDB Journal.* 1(1):1-39.

[Bre92c] BREITBART Y, GARCIA-MOLINA H & SILBERSCHATZ A. 1992. Overview of Multidatabase Transaction Management Scheduling. *VLDB Journal.* Vol 1, Number 2.

[Bre92d] BREITBART Y, GARCIA-MOLINA H & SILBERSCHATZ A. 1992. Overview of Multidatabase Transaction Management. *Technical Report TR-92-21,* Department of Computer Sciences, University of Texas at Austin, Austin, Texas 78712-1188.

[Bre95] BREITBART Y, GARCIA-MOLINA H & SILBERSCHATZ A. 1995. Transaction Management in Multidatabase Systems. In: *Modern Database Systems,* edited by: Won Kim. New York: Addison Wesley.

[Bri92] BRIGHT M W, HURSON A R & PAKZID S. 1992. A Taxonomy and Current Issues in Multidatabase Systems. *Computer.* 25(3):50-60.

[Buk93] BUKHRES O, CHEN J, DU W, ELMAGARMID A & PEZZOLI R. 1993. Interbase: An execution environment for global applications over distributed, heterogeneous, and autonomous software systems. *IEEE Computer.* August 1993. p57-69.

[Che93] CHEN J, BUKHRES O A & SHARIF-ASKARY J. 1993. A Customized Multidatabase Transaction Management Strategy. In: *4th International Conference, DEXA '93. Database and Expert Systems Applications,* edited by: Vladimír Mařik, Jiří Lažanský, Roland R Wagner. Prague, Czech Republic, September 1993 Proceedings.

[Cou94] COULOURIS G, DOLLIMORE J & KINDBERG T. 1994. *Distributed Systems: Concepts and Design.* Second Edition. Great Britain: Addison Wesley.

[Cri91] CRISTIAN F. 1991. Understanding Fault-Tolerant Distributed Systems. *Communications of the ACM.* Vol 34, No 2.

[Dat87] DATE C J. 1987. *A guide to The SQL standard.* Addison Wesley Publishing Company.

[Du89] DU W & ELMAGARMID A K. 1989. Quasi serializability: a correctness criterion for global concurrency control in InterBase. In: *Proceedings of the Fifteenth International Conference on Very Large Databases,* edited by P M G Apers & G Wiederhold. Palo Alto: Morgan Kaufmann. p347-355.

[Du91] DU W, ELMAGARMID A K & KIM W. 1991. Maintaining quasi serializability in multidatabase systems. In: *Proceedings of the Seventh International Conference on Data Engineering.* Kobe, Japan. p360-367.

[Du89] DU W, ELMAGARMID A K, LEU Y & OSTERMAN S. 1989. Effects of autonomy on global concurrency control in heterogeneous distributed database systems. In: *Proceedings of the Second International Conference on Data and Knowledge Systems for Manufacturing and Engineering.* Los Alamitos: IEEE Computer Society Press. p113-120.

[Elm86] ELMAGARMID A K & HELAL A. 1986. Heterogeneous database systems. Technical Report TR-86-004. The Pennsylvania State University, University Park, Pennsylvania 16802.

[Elm87] ELMAGARMID A K & LEU Y. 1987. An optimistic concurrency control algorithm for heterogeneous distributed database systems. *Q. Bull IEEE TC on Data Engineering.* 10(3):26-32.

[Elm88] ELMAGARMID A K & HELAL A. 1986. Supporting updates in heterogeneous distributed database systems. In: *Proceedings of the fourth International Conference on Data Engineering.* p564-569.

[Elm90a] ELMAGARMID A K & DU W. 1990. A paradigm for concurrency control in heterogeneous distributed database systems. In: *Proceedings of the Sixth International Conference on Data Engineering.* Los Alamitos: IEEE Compter Society. p37-46.

[Elm90b] ELMAGARMID A K , LEU Y, LITWIN W & RUSINKIEWICZ M. 1990. A multidatabase transaction model for Interbase. In: *Proceedings of the Sixteenth International Conference on Very Large Databases.* Brisbane.

[Els94] ELMASRI R & NAVATHE S B. 1994. *Fundamentals of Database Systems.* Benjamin/Cummings Publishing Company. Redwood City, California.

[Gar83] GARCIA-MOLINA H. June 1983. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems.* 8(2):186-213.

[Gar87] GARCIA-MOLINA H & SALEM K. 1987. Sagas. In: *Proceedings of ACM-SIGMOD. 1987 International Conference on Management of Data.* San Francisco. p249-259.

[Gar94] GARCIA-MOLINA H & KOGAN B. 1994. Node Autonomy in Distributed Systems. In: *Multidatabase Systems: An Advanced Solution for Global Information Sharing,* edited by A R Hurson, M W Bright & A Pakzad. Los Alamitos: IEEE Computer Society Press. Los Alamitos, California.

[Geo90] GEORGAKOPOULOS D. 1990. Transaction Management in Multidatabase Systems. Dissertation. University of Houston.

[Geo91a] GEORGAKOPOULOS D. 1991. Multidatabase Recovery and Recoverability. In: *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems,* edited by: Y. Kambayashi, M. Rusinkiewicz & A. Sheth. Kyoto, Japan. April 1991.

[Geo91b] GEORGAKOPOULOS D, RUSINKIEWICZ M & SHETH A P. April 1991. On Serializability of Multidatabase Transactions Through Forced Local Conflicts. In: *Proceedings of the 7th International Conference on Data Engineering.* Los Alamitos: IEEE Computer Society Press. p314-23.

[Geo94] GEORGAKOPOULOS D, RUSINKIEWICZ M & SHETH A P. 1994. Using Tickets to Enforce the Serializability of Mulitdatabase Transactions. *IEEE Transactions on Knowledge and Data Engineering.* 6(1):166-80.

[Goy91] GOYAL M L & SINGH G V. 1991. Access Control in Distributed Heterogeneous Database Management Systems. *Computers and Security.* 10(7):661-669.

[Gli84] GLIGOR V & LUCKENBAUGH G L. 1984. Interconnecting Heterogeneous Database Management Systems. *Computer* 17(1):33-43.

[Gli86] GLIGOR V & POPESCU-ZELETIN R. 1986. Transaction Management in Distributed Heterogeneous Database Management Systems. *Information Systems.* 11(4):287-297.

[Gra81] GRAY J N. 1981. The transaction concept. In: *Proceedings of the Seventh International Conference on Very Large Databases.* Cannes. p144-154.

[Gra93] GRAY J & REUTER A. 1993. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, San Francisco, California.

[Had88] HADZILACOS V. 1988. A Theory of Reliability in database systems. *Journal of the ACM.* 35(1):121-145.

[Her90] HERLIHY M. 1990. Apologizing versus asking permission: optimistic concurrency control for abstract data types. *ACM Transactions on Database Systems.* 15(1):96-124.

[Hsa92] HSAIO D. 1992. Tutorial on federated databases and systems (part 1). *VLDB Journal.* 1(1).

[Hwa94] HWANG S, SRIVASTAVA J & LI J. 1994. Transaction Recovery in Federated Autonomous Databases. *Distributed and Parallel Databases 2.* 2(2).

[Jin93] JIN W W, RUSINKIEWICZ M, NESS L & SHETH A. 1993. Concurrency Control and Recovery of Multidatabase Work Flows in Telecommunication Applications. *SIGMOD Record.* 22(2):456-459.

[Kam92] KAMEL M N & KAMEL N N. 1992. Federated database management system: Requirements, issues and solutions. *Computer Communications.* 15(4):270-277.

[Kan93] KANG I E & KEEFE T F. 1993. Supporting reliable and atomic transaction management in multidatabase systems. In: *Proceedings of the 13th International Conference on Distributed Computing Systems.* Los Alamitos: IEEE Computer Society Press. p457-464.

[Key93] KEYSER P W. 1993. Transaction Management in Multidatabase Management Systems. In: *Proceedings of the 8th National Conference for Masters and PhD students in Computer Science.* UNISA, Pretoria, South Africa.

[Kim92] KIM Y S. 1992. Atomic Transaction Scheduling in Tightly Coupled Heterogeneous Distributed Databases. Ph.D. Thesis, Department of Information and Communications Engineering, Korea Advanced Institute of Science and Technology, Seoul, Korea.

[Kim93] KIM P C, KIM W & SILBERSCHATZ A. 1993. Concurrency Control and Recovery in multidatabase systems. *International Journal of Computer and Software Engineering.*

[Kim94] KIM K H. 1994. Fair distribution of concerns in design and evaluation of fault-tolerant distributed computer systems. *Computer Communications.* 17(10):699-707.

[Kor83] KORTH H F. January 1983. Locking primitives in a database system. *Journal of the ACM.* 30(1):55-79.

[Kor88] KORTH H F, KIM W & BANCILHON F. 1988. On long duration CAD transactions. *Information Sciences.* 46:73-107.

[Kor90] KORTH H F, LEVY E & SILBERSCHATZ A. August 1990. A formal approach to recovery by compensating transactions. In: *Proceedings of the Sixteenth International Conference on Very Large Databases,* edited by D McLeod, R Sacks-Davis & H Schek. Palo Alto: Morgan Kaufmann. p95-106.

[Küh94] KÜHN E. 1994. Fault-Tolerance for Communicating Multidatabase Transactions. In: *Proceeding of the 27th Hawaii International Conference on System Sciences. Vol II: Software Technology,* edited by E R Hesham & B D Shriver. Los Alamitos: IEEE Computer Society Press. p323-332.

[Lag90] *Database systems: Achievements and opportunities,* 1990. The Lagunita Report of the NSF Invitational Workshop on the Future of Database Systems Research. Palo Alto, California, February 1990, edited by: Avi Silberschatz, Michael Stonebraker & Jeffrey Ullman.

[Lev91a] LEVY E, KORTH H F & SILBERSCHATZ A. May 1991. An optimistic commit protocol for distributed transaction management. *SIGMOD Record.* 20(2):88-97.

[Lev91b] LEVY E, KORTH H F & SILBERSCHATZ A. August 1991. A theory of relaxed atomicity. In: *Proceedings of the Tenth annual ACM Symposium on Principles of Distributed Computing.* New York: ACM. p95-109.

[Lit89] LITWIN W & TIRRI H. 1989. Flexible concurrency control using value dates. In: *Integration of Information Systems: Bridging Heterogeneous Database,* edited by: A. Gupta. New York: IEEE Press. p144-149.

[Lyn83] LYNCH N. December 1983. Multi-level atomicity. *ACM Transactions on Database Systems.* 8(4):484-502.

[Meh91] MEHROTRA S, RASTOGI R, KORTH H F & SILBERSCHATZ A. 1991. Nonserializable executions in heterogeneous distributed database systems. In: *Proceedings of the First International Conference on Parallel and Distributed Systems.* Los Alamitos: IEEE Computer Soceity Press. p245-252.

[Meh92a] MEHROTRA S, RASTOGI R, BREITBART Y, KORTH H F & SILBERSCHATZ A. 1992. The Concurrency Control Problem in Multidatabases: Characteristics and Solutions. In: *ACM SIGMOD.* 21(2): 288-297.

[Meh92b] MEHROTRA S, RASTOGI R, KORTH H F & SILBERSCHATZ A. 1992. A Transaction Model for Multidatabase Systems. In: *Proceedings of the 12th International Conference on Distributed Computing Systems.* Los Alamitos: IEEE Computer Society Press. p56-63.

[Meh92c] MEHROTRA S, RASTOGI R, BREITBART Y, KORTH H F & SILBERSCHATZ A. 1992. Ensuring Transaction Atomicity in Multidatabase Systems. In: *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems.* Baltimore: ACM Press. p164-175.

[Meh92d] MEHROTRA S, RASTOGI R, KORTH H F & SILBERSCHATZ A. 1992. Relaxing Serializability in Multidatabase Systems. In: *Second International Workshop on Research Issues on Data Engineering: Transaction and Query Processing,* edited by Philip S Yu. Los Alamitos: IEEE Computer Society Press. p205-212.

[Meh93] MEHROTRA S. 1993. Failure-Resilient Transaction Management in Multidatabase Systems. Doctor of Philosophy Dissertation. University of Texas at Austin.

[Mul92] MULLEN J G, ELMAGARMID A K, KIM W & SHARIF-ASKARY J. 1992. On the Impossibility of Atomic Commitment in Multidatabase Systems. In: *Proceedings of The Second International Conference on Systems Integration.* edited by: P A Ng, C V Ramamoorthy, L C Seifert & R T Yeh. Los Alamitos: IEEE Computer Society Press. p625-634.

[Mul93] MULLEN J G, JING J & SHARIF-ASKARY J. 1993. Reservation Commitment and Its Use in Multidatabase Systems. In: *4th International Conference, DEXA '93. Database and Expert Systems Applications,* edited by: Vladimir Mařik, Jiři Lažanskỳ, Roland R Wagner. Prague, Czech Republic.

[Mut91] MUTH P & RAKOW T C. 1991. Atomic Commitment for Integrated Database Systems. In: *Proceedings of the 7th International Conference on Data Engineering.* Los Alamitos: IEEE Computer Society Press. p296-304.

[Nam93] NAM H & MOON S. 1993. Global Deadlock Detection for Concurrency Control in Multidatabase Systems. *Microprocessing and Microprogramming.* 39:155-158.

[Nod93] NODINE M H. 1993. Supporting long-running tasks on an evolving multidatabase using interactions and events. In: *Proceedings of the Second International Conference on Parallel and Distributed Information Systems.* Los Alamitos: IEEE Computer Society Press. p125-132.

[Nod94] NODINE M H. 1994. Automating compensation in a multidatabase. In: *Proceedings of the Twenty-Seventh Hawaii International Conference on Systems Sciences. Vol II Software Technology,* edited by E R Hesham & B D Shriver. Los Alamitos: IEEE Computer Society Press. p292-312.

[Özs90] ÖZSU M T & BARKER K. 1990. Architectural Classification and Transaction Execution Models of Multidatabase Systems. *Lecture Notes in Computer Science.* 468(5):285-294.

[Özs91] ÖZSU M T & VALDURIEZ P. 1991. *Principles of Distributed Database Systems.* Englewood Cliffs, New Jersey. Prentice Hall.

[Özs94] ÖZSU M T. Distributed Databases. In: *Readings in Distributed Computer Systems,* edited by: T L Casavant & M Singhal. Los Alamitos: IEEE Computer Society Press.

[Per91] PERRIZO W, RAJKUMAR J & RAM P. 1991. Hydro: A heterogeneous distributed database system. *SIGMOD Record.* 20(2):32-39.

[Pu88] PU C. 1988. Superdatabases for composition of heterogeneous databases. In: *Multidatabase Systems: An Advanced Solution for Global Information Sharing,* edited by A R Hurson, M W Bright & A Pakzad. Los Alamitos: IEEE Computer Society Press. Los Alamitos, California.

[Pu91a] PU C & LEFF A. 1991. Replica control in distributed systems. *SIGMOD Record.* 20(2):377-386.

[Pu91b] PU C, LEFF A & CHEN S F. 1991. Heterogeneous and Autonomous Transaction Processing. *Computer.* Dec 1991. p64-72.

[Ras92] RASTOGI R, KORTH H F & SILBERSCHATZ A. 1992. Exploiting transaction semantics in multidatabase systems. *Technical Report TR-92-45,* Department of Computer Science, University of Texas at Austin.

[Ras93a] RASTOGI R, KORTH H F & SILBERSCHATZ A. 1993. Strict histories in object-based database systems. In: *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems.* Washington D.C.

[Ras93b] RASTOGI R R. 1993. Concurrency Control in Multidatabase Systems. PhD Thesis. University of Texas.

[Raz92] RAZ Y. 1992. The principle of atomic commitment ordering, or guaranteeing serializability in a heterogeneous environment of multiple autonomous resource managers using atomic commitment. In: *Proceedings of the Eighteenth International Comference on Very Large Databases.* Vancouver. p292-312.

[Rus90] RUSINKIEWICZ M, ELMAGARMID A, LEU Y, & LITWIN W. 1990. Extending the transaction model to capture more meaning. *SIGMOD record.* 19(1).

[Rus92] RUSINKIEWICZ M, KRYCHNIAK P & CICHOCKI A. December 1992. Towards a Model for Multidatabase Transactions. *International Journal of Intelligent and Cooperative Information Systems.* 1(3-4):579-617.

[Sal89] SALEM K, GARCIA-MOLINA H & ALONSO R. 1989. Altruistic Locking: A strategy for coping with long-lived transactions. *Lecture Notes in Computer Science, High Performance Transaction Processing Systems*, edited by: D Gawlick, M Haynie, A Reuter. Vol 359, Springer Verlag, p175-199.

[Sha92] SHASHA D, SIMON E & VALDURIEZ P. 1992. Simple relational guidance for chopping up transactions. In: *Proceedings of ACM-SIGMOD 1992 Conference on Management of Data.* San Diego, California, p298-307.

[Sil83] SILBERSCHATZ A, STONEBRAKER M & ULLMAN J. 1983. Database Systems: Achievements and Opportunities. In: *Readings in Database Systems, Second Edition,* edited by Michael Stonebraker. Morgan Kaufmann Publishers, San Francisco, California. 1994.

[Sop91a] SOPARKAR N R, KORTH H F & SILBERSCHATZ A. 1991. Trading control autonomy for reliability in Multidatabase Transactions. *Technical Report TR-91-05.* The University of Texas at Austin, Computer Sciences Department.

[Sop91b] SOPARKAR N R, KORTH H F & SILBERSCHATZ A. 1991. Failure-Resilient Transaction Management in Multidatabases. *Computer.* Dec 1991. p28-36.

[Sto94] STONEBRAKER M, WONG E & KREPS P. 1994. The Design and Implementation of Ingres. In: *Readings in Database Systems, Second Edition,* edited by Michael Stonebraker. Morgan Kaufmann Publishers, San Francisco, California.

[Sug87] SUGIHARA K. 1987. Concurrency Control Based on Distributed Cycle Detection. In: *IEEE Prooceeding of the 3rd International Conference on Data Engineering, Los Angeles, California.* U.S.A. p 267-274.

[Tan93] TANG X. 1993. Multidatabase Transaction Management: A Study of the 2PC Agent Method. MSc Thesis. University of Waterloo.

[Tun92] TUNG H. 1992. Deadlock Detection and Resolution in Distributed Database Systems and Multidatabase Systems. PhD Thesis. Northwestern University.

[Vei89] VEIJALAINEN J. 1989. *Transaction Concepts in Autonomous Database Environments.* R. Olderbourg Verlag, Munich.

[Vei92] VEIJALAINEN J & WOLSKI A. 1992. Prepare and commit certification for decentralized transaction management in rigorous heterogeneous multidatabases. In: *Proceedings of the Eighth International conference on Data Engineering.* Los Alamitos: IEEE Computer Society Press. p470-479.

[Ver78] VERHOFSTAD J S M. 1978. Recovery Techniques For Database Systems. *ACM Computing Surveys.* 10(2):167-196.

[Vid91] VIDYASANKAR K. 1991. A Non-Two-Phase Locking Protocol for Global Concurrency Control in Distributed Heterogeneous Database Systems. *IEEE Transactions on Knowledge and Data Engineering.* Vol 3, No 2. pp 256-260.

[Wei88] WEIHL W E. 1988. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488-1505.

[Wei89] WEIHL W E. 1989. The impact of recovery on concurrency control. In: *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems.* New York: ACM. p259-269.

[Wol90] WOLSKI A & VEIJALAINEN J. 1990. 2PC agent method. Achieving serializability in the presence of failures in a heterogeneous multidatabase. In: *Proceedings of the International conference on databases, parallel architectures and their applications*, edited by N Rishe, S Navathe & D Tal. Los Alamitos: IEEE Computer Society Press. p321-330.

[Won92] WONG M H & AGRAWAL D. June 1992. Tolerating bounded inconsistency for increasing concurrency in database systems. In: *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems.* Baltimore: ACM Press. p236-245.

[Wu92] WU K L, YU P S, PU C. 1992. Divergence control for epsilon-serializability. In: *Proceedings of the Eighth International Conference on Data Engineering*, Tempe, Arizona. p506-515.

[Ye94] YE X & KEANE J A. 1994. A Distributed Transaction Management Scheme for Multidatabase Systems. In: *Proceedings of the 1994 IEEE Region 10's Ninth Annual International Conference. Theme: Frontiers of Computer Technology*, Singapore. p397-399.

[Yoo95] YOO H, KIM M H. 1995. A reliable Global Atomic Commitment Protocol for Distributed Multidatabase Systems. *Information Sciences.* 83(1-2):49-76.

[Yun93] YUN H & HWANG B. 1993. A Pessimistic Concurrency Control Algorithm in Multidatabase Systems. In: *Database Systems for Advanced Applications '93. Proceedings of the Third International Symposium on Database Systems for Advanced Applications. Taejoon, South Korea*, edited by: S Moon & H Ikeda. Singapore: World Scientific. p379-386.

[Zha93] ZHANG Y & ORLOWSKA M E. 1993. A Hybrid Concurrency Control Approach in Heterogeneous Distributed Database Systems. In: *Proceedings TENCON '93, 1993 IEEE Region 10 Conference on Computer, Communications, Control and Power Engineering*, edited by Yuan Baozong. Vol 1. New York: IEEE. p323-326.