A categorization scheme for concurrency control protocols in distributed databases

Yunyong Teng-amnuay

Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Part of the Computer Sciences Commons

Recommended Citation
https://lib.dr.iastate.edu/rtd/7734

This Dissertation is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify markings or notations which may appear on this reproduction.

1. The sign or “target” for pages apparently lacking from the document photographed is “Missing Page(s)”. If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure complete continuity.

2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed, a definite method of “sectioning” the material has been followed. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.
Teng-Amnuay, Yunyong

A CATEGORIZATION SCHEME FOR CONCURRENCY CONTROL PROTOCOLS IN DISTRIBUTED DATABASES

Iowa State University

University Microfilms International 300 N. Zeeb Road, Ann Arbor, MI 48106

Ph.D. 1984
A categorization scheme for concurrency control protocols in distributed databases

by

Yunyong Teng-amnuay

A Dissertation Submitted to the Graduate Faculty in Partial Fulfillment of the Requirements for the Degree of DOCTOR OF PHILOSOPHY

Major: Computer Science

Approved:

In Charge of Major Work

For the Major Department

For the Graduate College

Iowa State University
Ames, Iowa
1984
# TABLE OF CONTENTS

I. INTRODUCTION ........................................... 1

A. Concurrency Control Problem ........................... 1
   1. Consistency of shared resources .................. 2
   2. Concurrency control in a centralized database ... 6
      a. Definition ..................................... 6
      b. Philosophy of concurrency control ............. 9
   3. Concurrency control in a distributed database .... 12
      a. Definition ..................................... 12
      b. Advantages of a distributed database .......... 14
      c. Difficulties inherent in a distributed database ... 15
      d. Common synchronization techniques ............... 17
         1) Locking ...................................... 17
         2) Timestamping .................................. 18
      e. Mutual consistency ................................ 19

B. Essence of Research ..................................... 19
   1. Problem statement .................................. 19
   2. Review of previous categorization schemes ........ 21
   3. Purpose of the research ............................ 24
      a. Development of categorization scheme .......... 24
      b. Synthesis of concurrency control algorithms ... 25
   4. Benefits of the research ............................ 26
      a. Benefit due to the categorization scheme ...... 26
      b. Benefit due to synthesis of algorithms ......... 28
   5. Research outline .................................... 29

II. SURVEY OF EXISTING CONCURRENCY CONTROL METHODS .... 31

A. Description of Methods .................................. 31
   1. Unanimous consensus with full redundancy .......... 33
   2. Action processing in timestamp order .............. 34
   3. SDD-1 concurrency control scheme .................. 35
   4. Distributed data-sharing system using tickets ... 38
   5. Distributed locking ................................ 40
   6. Inconsistency avoidance in daisy-chained system ... 42
   7. Primary-copy locking scheme ........................ 43
   8. Majority consensus with full redundancy .......... 45
   9. Distributed precedence-relation analysis .......... 47
   10. Centralized control with full redundancy ........ 48
   11. Distributed request queue with full
12. Closely synchronized distributed clock. 51
13. Reversible unanimous consensus. 52
14. Migrating central control by unanimous consensus. 53
15. Distributed waiting queue with full redundancy. 54
16. Centralized serializability controller. 55
17. Distributed reservation list with full redundancy. 56
18. Migrating control on virtual ring. 58
19. Deadlock-free resource management with ordered locking of data. 59
20. Multiversion database. 60

B. Summary. 61

III. FRAMEWORK ON PROBLEM OF CONCURRENCY CONTROL. 62

A. Causes of Inconsistency. 62
   1. Cause of inconsistency due to erroneous inputs. 64
   2. Cause of inconsistency due to concurrent execution. 64
   3. Cause of inconsistency due to hardware. 65

B. Environment for the Control of Concurrency. 65
   1. Organization of data. 66
   2. Data dictionary. 68
   3. Physical network architecture. 70
   4. Transaction model. 70
      a. Collapsing the complex transaction. 71
      b. Multi-traced transaction. 74
      c. Intra-dependent transaction. 76

C. Basic Distributed Database Model. 78
   1. Components of the BDD model. 79
   2. Logical components. 79
      a. Transaction manager. 79
      b. Data manager. 81
      c. Data network. 82
      d. Control network. 82
   3. Transaction life-cycle. 83

D. Summary. 86

IV. DISCUSSION ON SERIALIZABLE EXECUTION SEQUENCES
   AND THEORETICAL BASIS FOR THE CATEGORIZATION SCHEME. 87

A. Serializability Concept. 88
B. Aspects of Classes of Serializable Execution

Sequences ............................. 95
1. Equivalence of DSR, CPSR, and CG-secure .... 97
2. SDD-1-secure is a subset of P3 .......... 99
3. Equivalence of classes Q and W2PL .... 100
4. Implications of the classes CPSR and SSR .. 103
   a. Notions of liveness, deadness and
detailed transactions .................. 104
   b. Subclasses of SR concerning live actions . 108
   c. On the possibly inefficient recognition
of members of SSR ..................... 110
5. CPSR class as a basis for categorization .. 113

C. Summary ........................... 113

V. THE CATEGORIZATION SCHEME ............... 114

A. A Pragmatic Reason for Selecting CPSR Class .. 114

B. Philosophy of Primary Categories ............ 118
   1. Logical approaches to concurrency control
      problem ................................ 119
      a. Prevention of nonserializability ..... 119
      b. Correction of nonserializability ..... 120
   2. The role of transaction rejection ........ 120
   3. Distinction between subcategories and
      components .......................... 121

C. Subcategorization of Prevention ............. 122
   1. Preanalyzer .......................... 122
      a. Organization of simplified preanalyzer . 123
         1) Centralized simplified preanalyzer .. 124
         2) Distributed simplified preanalyzer .. 125
      b. Provision for concurrency ............ 125
         1) Dynamic conflict-analysis .......... 126
         2) Static conflict-analysis .......... 127
   2. Enforcer ............................ 127
      a. OAS with consecutive timestamps .... 128
      b. OAS with nonconsecutive timestamps .. 129
         1) OAS for read actions ............. 130
         2) OAS for write actions ............ 130
         3) Reservation of write ............. 131
         4) Provision for local concurrency ... 132
   3. Categorization tree for prevention ...... 133

D. Subcategorization of Correction ............. 133
   1. Integrity of conflict-analysis .......... 135
      a. Integral conflict-analysis .......... 135
         1) Multiplicity of integral analyzer .. 136
E. Categorization of Some Existing Concurrency Control Algorithms

F. Summary

VI. SYNTHESIS OF CONCURRENCY CONTROL ALGORITHMS

A. Selection of Classes

1. CPSR class
2. Class Q
3. Class 2PL
4. Other classes

B. Synthesis of Centralized Concurrency Control Algorithms

1. Three-level precedence graph
   a. Shortcomings of a conflict-graph and a precedence relation
   b. Definition of the 3PG
2. Synthesis of centralized concurrency control algorithms producing the class CPSR
   a. Centralized preventive CPSR algorithm
   b. Centralized corrective CPSR algorithm
3. Synthesis of centralized concurrency control algorithms producing the class Q
   a. Centralized preventive Q algorithm
   b. Centralized corrective Q algorithm
4. Synthesis of centralized concurrency control algorithm producing the class 2PL

C. Synthesis of Distributed Concurrency Control Algorithms

1. A distributed preventive CPSR algorithm
   a. Sequencer
   b. TM
   c. DM
   d. The complex waiting queue
      1) Subqueues
      2) Batch of data dependencies
      3) Detailed action handling
   e. Algorithm for the complex waiting queue
f. An example of the complex waiting queue . 196
2. Distributed preventive Q\(^\prime\) algorithms . . . . 200
   a. Simple preventive Q\(^\prime\) (SPQ) algorithm . . 202
   b. Virtual ordering . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 203
   c. Complex preventive Q\(^\prime\) (CPQ) algorithm . . 204
3. Distributed preventive 2PL algorithms . . . . 206
   a. Synchronization technique for preventive
      2PL algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 207
   b. Simple preventive 2PL (SP2PL) algorithm . . 208
   c. Complex preventive 2PL (CP2PL) algorithm . . 210
D. Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 211
VII. CONCLUSION . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 212
   A. Research Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 212
   B. Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 213
   C. Further Research . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 214
VIII. BIBLIOGRAPHY . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 216
LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Subcategories for Prevention</td>
<td>134</td>
</tr>
<tr>
<td>5.2</td>
<td>Subcategories for Correction</td>
<td>145</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>Title</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Model of a centralized database system</td>
<td>9</td>
</tr>
<tr>
<td>1.2</td>
<td>Model of a site in a distributed database system</td>
<td>13</td>
</tr>
<tr>
<td>3.1</td>
<td>A Basic Distributed Database Model</td>
<td>80</td>
</tr>
<tr>
<td>4.1</td>
<td>An example of the conflict-graph</td>
<td>95</td>
</tr>
<tr>
<td>4.2</td>
<td>Allowable execution sequences of classes Q and WZPL</td>
<td>104</td>
</tr>
<tr>
<td>4.3</td>
<td>Relationship among subclasses of serializable execution sequences</td>
<td>105</td>
</tr>
<tr>
<td>4.4</td>
<td>Relationship among some selected classes in SR</td>
<td>111</td>
</tr>
<tr>
<td>5.1</td>
<td>Left-to-right parsing for a name of an algorithm</td>
<td>147</td>
</tr>
<tr>
<td>5.2</td>
<td>Interpretation of an example of category names</td>
<td>148</td>
</tr>
<tr>
<td>6.1</td>
<td>Conflicting edges in CG between two transactions</td>
<td>160</td>
</tr>
<tr>
<td>6.2</td>
<td>CG for complex transactions</td>
<td>160</td>
</tr>
<tr>
<td>6.3</td>
<td>CG for ES with an imposed direction on arcs</td>
<td>162</td>
</tr>
<tr>
<td>6.4</td>
<td>CG for ES based on transaction as a whole with an imposed direction on arcs</td>
<td>162</td>
</tr>
<tr>
<td>6.5</td>
<td>An example of the first-level 3PG</td>
<td>164</td>
</tr>
<tr>
<td>6.6</td>
<td>An example of the second-level 3PG</td>
<td>165</td>
</tr>
<tr>
<td>6.7</td>
<td>An example of the third-level 3PG</td>
<td>165</td>
</tr>
<tr>
<td>6.8</td>
<td>An example of the simplification of a</td>
<td></td>
</tr>
</tbody>
</table>
third-level 3PG

FIGURE 6.9. Another example of the simplification of a third-level 3PG

FIGURE 6.10. Simplified third-level 3PG

FIGURE 6.11. Second-level 3PG for ES₄

FIGURE 6.12. First-level 3PG for ES₄, showing a directed cycle

FIGURE 6.13. Third-level 3PG for a preventive Q algorithm

FIGURE 6.14. A snapshot of the waiting queue at DM₁

FIGURE 6.15. Subqueues of the consecutive set in Figure 6.14

FIGURE 6.16. Synchronization requirements for 2PL algorithm
I. INTRODUCTION

The main purpose of this research is to form a taxonomy of various methods for the control of concurrency in distributed databases. In this chapter, the problem of concurrency control is described and a review of previous classification attempts is given. This review, along with subsequent discussion, establishes the need for the research in this dissertation. At the end of this chapter, a brief outline of the dissertation is presented.

A. Concurrency Control Problem

In recent years, distributed computing has enjoyed tremendous growth. This is due to the appealing nature of the concept and the explosive advances in computing hardware. The distribution of data, over geographically separated sites, has introduced numerous new problems which require solutions. In particular, the need for concurrency control in distributed databases is a natural outgrowth of this development. The present state of this area of research has been expressed by Bernstein and Goodman (1981):

Distributed concurrency control is in a state of extreme turbulence. More than twenty concurrency control algorithms have been proposed for distributed databases, and several have been, or are being, implemented. These algorithms are usually complex, hard to understand, and difficult to prove correct (indeed, many are incorrect). Because they are described in different terminologies and make different assumptions about
the underlying distributed database environment, it is difficult to compare the many proposed algorithms, even in qualitative terms. Naturally each author proclaims his or her approach as best, but there is little compelling evidence to support the claims.

In order to understand the nature of this research, this chapter lays out the underlying need for concurrency control. This chapter also attempts to establish a working terminology of certain common concepts. Although the definitions and concepts are selected from various contributions to the literature, they cannot conceivably cover every possible variation. Nevertheless, the attempt to derive suitable definitions should serve as an adequate guideline and also convey the feeling of the turbulent state of the field, as observed by Bernstein and Goodman.

1. Consistency of shared resources

In a computing system, there is a need to execute more than one task at any given time. This is due to the prohibitive cost and expense of the computer and also to the mismatch in speeds between computers and human beings. If a task is started before another task has finished, then the two tasks are said to overlap and the execution of the two tasks is said to be concurrent. Individual steps inside each of the two concurrent tasks have to be interleaved if the system has only one processing unit.

When concurrent tasks share some common resource, there
are interactions among the tasks through the state of the shared resource. These interactions, if not properly controlled, can violate the consistency of the shared resource. The consistency of the resource is defined by inherent relationships or predicates, which may be impossible or impractical to enumerate. Such predicates are called consistency constraints. To illustrate the concept of consistency, consider resources in the form of two banking accounts: savings (S) and checking (C). The consistency constraint is that the balance of each account must reflect deposits and withdrawals on the account. Let $T_1$ and $T_2$ be two tasks which may concurrently use the two accounts. The following are the internal details of the two tasks.

$$T_1: \text{begin} \quad /*\text{transfer from savings to checking} */$$

\begin{align*}
&\text{fetch the balance of } S; \\
&S = S - (\text{amount_of_transfer}); \\
&\text{store the balance of } S; \\
&\text{fetch the balance of } C; \\
&C = C + (\text{amount_of_transfer}); \\
&\text{store the balance of } C
\end{align*}

$$\text{end.}$$
$T_2: \text{ begin } /* \text{ deposit to savings } */$

fetch the balance of $S$;

$S = S + (\text{amount\_of\_deposit});$

store the balance of $S$

end.

Let $R$ and $W$ represent the fetch (read) and store (write) operations, respectively. Then, $T_1$ can be represented by

$$R_1(S)W_1(S)R_1(C)W_1(C).$$

Similarly, $T_2$ can be represented by

$$R_2(S)W_2(S).$$

If the two transactions are processed sequentially (serially) either as

$$R_1(S)W_1(S)R_1(C)W_1(C)R_2(S)W_2(S)$$

or

$$R_2(S)W_2(S)R_1(S)W_1(S)R_1(C)W_1(C)$$

then, the consistency is preserved. The interleaved sequence

$$R_1(S)W_1(S)R_2(S)W_2(S)R_1(C)W_1(C)$$

also preserves consistency. Note that this sequence produces the same result as that of the first of the above
serial sequences. On the other hand, suppose the steps of $T_1$ and $T_2$ are interleaved in the following manner:

$$R_1(S)R_2(S)W_2(S)W_1(S)R_1(C)W_1(C).$$

In this case, the consistency is violated since the final balance of the saving account will reflect only the transfer.

Usually, it is assumed that any task, when executed alone (nonconcurrent with any other task), will preserve the consistency of resources. The above example implies that inconsistency can arise when tasks interfere with each other in accessing common resources. Thus, if a task is indivisible or is given the appearance of being so, then consistency is preserved. A task is **indivisible** or **atomic** (Davies et al., 1981) if

1. either all steps comprising the task are completed or no step is completed, and
2. intermediate states or results are not visible to any other task.

Intuitively, indivisibility means that the effect of the task on shared resources appears as though the whole task has been performed in a single step.
2. **Concurrency control in a centralized database**

A shared resource in a computing system can be anything from the smallest binary digit to the whole computing system itself. The problem of a single shared resource has been extensively researched for conventional operating systems. A database, on the other hand, represents a unique class of shared resources which cannot be effectively handled by conventional means. The shared resources, in this case, are the data defined by the users. This section presents definitions of a centralized database, a transaction system and related concepts. The philosophy of concurrency controls is also discussed.

a. **Definition**

Formally (Bernstein et al., 1979), a database \( D \) is a set of distinct data \( \{X_1, \ldots, X_m\} \). The granularity of \( D \) can vary depending on specific systems. The domain of a datum is represented as \( \text{dom}(X_i), i=1, \ldots, m \). Hence, a database state is an element of the domain of the database state, \( \text{dom}(D) \), whose definition is

\[
\text{dom}(D) = \text{dom}(X_1) \times \text{dom}(X_2) \times \ldots \times \text{dom}(X_m).
\]

There are two classes of data. The data, as seen by the user, forms the user database. The other class, called the administrative database, constitutes all other information brought into existence by the need to manage the data. In this work, maintaining consistency of the user
database is the concern. Concurrency control algorithms ensure the consistency of the user database through the information it has stored in the administrative database.

In a database, a task takes the form of a transaction which is a unit of work deemed indivisible by the user and hence preserves consistency when executed alone. A transaction consists of the execution of an application-specified sequence of operations. A transaction can be considered an interface between a database and its users.

A transaction affects the state of a database through interactions (usually called database actions or simply actions). There are two types of database actions. The first type is used to change the domain of the database, i.e. to create and destroy data. The other type of action concerns the usage, or access, of data, i.e. read and write actions. A read action acquires the data required by the transaction for its computation. Such data, called the read-set, may need more than one read action depending on the nature of the transaction. A write action, or update, changes the state of the database with the value(s) of data produced by the transaction. Again, more than one write action may be required. The portion of the database changed by a transaction is the transaction's write-set. The process in which updates are applied to the database may vary. It may consist of many phases which tentatively
affect the database in steps to increase reliability. A transaction is assumed to halt for all inputs within a finite period of time (usually, a bound is specified to ensure this). A termination may signal the successful processing of a transaction. In such a case, the transaction is said to be committed, i.e., the database state is irreversibly changed. A termination may also signify unsuccessful processing. In such a case, the transaction is said to be rejected (aborted, rolled back, restarted or preempted). A rejection may be a result of some exceptional condition, e.g. a needed datum cannot be found. It is a common assumption that a rejected transaction does not change the database state. A program may consist of more than one transaction. Usually, however, a program represents only one transaction.

Two transactions $T_i$ and $T_j$ are said to be in conflict (i.e. they interfere with each other), if either

- $(\text{read-set}_i \cup \text{write-set}_i) \cap \text{write-set}_j \neq \emptyset$, or
- $\text{write-set}_i \cap (\text{read-set}_j \cup \text{write-set}_j) \neq \emptyset$.

Otherwise, $T_i$ and $T_j$ are said to be nonconflicting. Two actions are said to be in conflicting or interfere with each other if they use at least one common datum and if at least one of the actions is a write action.

The entity that houses and manages data, and serves transactions is usually called a database system. A simple
model of a centralized database system is shown in Figure 1.1.

FIGURE 1.1. Model of a centralized database system

The term "data" in Figure 1.1 includes physical housing, data structures and their access methods. The database management system (DBMS) or database manager (DBM) translates transactions desired by users (or application programs) into specific lower level access commands. The control of concurrency is also a function of the DBMS if concurrency is allowed by the system.

b. Philosophy of concurrency control

Concurrency control constrains concurrent execution so that consistency is preserved. Concurrency control permits users to access a database in a concurrent fashion while preserving the illusion that each user is executing atomically.
In the conventional approach to concurrency control, a shared resource may be fenced off into a critical region where uses must be sequentially applied. Semaphores and monitors are used to lock such a region to guarantee exclusive or indivisible access to the resource. If the whole database is considered a single critical region, then all tasks using the database will have to be executed sequentially, and concurrency is nonexistent. One way to increase the degree of concurrency is to provide for granularity of the database. In this case, each transaction gains exclusive access only to the needed data. Transactions, which do not access common data, can be executed concurrently because they do not interfere with each other. Even nonconflicting transactions, the read-sets of which may overlap, can also be executed concurrently. The shortcoming of holding exclusive access for the entire life-time of a transaction is that any pair of conflicting transactions must be executed sequentially. This results in a lower degree of concurrency. Furthermore, the more finely divided the database is, the more numerous are the independent units of accessible data. In this case, the probability that any two transactions will conflict is lowered. However, the overhead of keeping track of the separate addressable units becomes more pronounced. In such a case, some appropriate tradeoffs must be made.
A logical extension of the above solution is to allow a transaction to gain exclusive access to its data in bits and pieces instead of reserving them exclusively as its own for the duration of its lifetime. Although this increases the degree of concurrency in the system, the breakdown of the indivisibility of a transaction can result in the inconsistency of the database. There are different ways to approach the problem. One widely accepted technique is the preservation of consistency based on serializability, the concept of which is briefly described here and will be addressed formally in Chapter IV. An interleaved execution of individual operations from various transactions is termed an execution sequence or a log. A serial execution sequence is any sequence in which transactions are processed sequentially (serially, one after another or nonconcurrently). Since each transaction is assumed to preserve consistency, it is obvious that a serial execution sequence will preserve consistency. Two execution sequences are said to be equivalent if, starting with the same initial database state, the two sequences produce identical resulting database states. Therefore, an execution sequence preserves the consistency of the database if it is equivalent to some serial execution sequence of the same set of transactions. Serializability is only a sufficient condition for consistency. Nevertheless, the concept is
appealing due to its simplicity.

3. **Concurrency control in a distributed database**

   The problem of preserving consistency is more difficult in a distributed database system. This section presents some common definitions of a distributed database and related concepts. The advantages and difficulties of a distributed database are also addressed.

   a. **Definition**  
      A working definition of a distributed database has been given by Date (1983). A distributed database contains data which are not stored in their entirety at a single physical location, but rather are spreaded across a network of locations (called sites or nodes) that are geographically dispersed and connected via communication links. It is important to note that a distributed database can be divided into distinct pieces such that, for a given user, access to some of those pieces is very much slower than access to others. Figure 1.2 depicts a general model of a site which is part of some distributed database system.

      The additions to the model in Figure 1.1 are the distributed database management system (DDBMS) and the communication network. Usually, a transaction is a part of some application program local to one of the database sites. Thus, in view of the rest of the system, this site is the originating or initiating site of the transaction. A DDBMS
can be considered a higher level software system which maps transactions' actions into local and remote actions. Local actions are serviced by the DBMS of the local database. Remote actions are forwarded as messages through the communication network to other appropriate DDBMSs.

It should be noted that the term "action" can be referred to in two contexts. First, an action is the dynamic process of performing some specified piece of work. In this sense, it is comprised of several components: the read-set and write-set which denote the objects acted upon
along with the portion of software code in the DDBMS and DBMS that implements the accessing of these objects.

Second, an action is viewed as a static piece of information denoting a request for service and which can be manipulated by various components of the system, e.g. an action can be sent as part of a message between two sites. In this latter case, it will be convenient to say that an action is sent from one site to another. Also, it will be convenient to say that a transaction accesses data. This is intended to mean that the actions of the transaction are sent and performed at some site(s).

b. Advantages of a distributed database There are a number of reasons why a distributed database is more attractive than a centralized one (Date, 1983 and Rothnie and Goodman, 1977).

1. Local autonomy: An enterprise using a database system is usually divided both logically and physically into groups of resources. Distributing the database allows individual groups to exercise local control over their own data, and thus are not dependent on some remote facilities to take care of purely local issues.

2. Incremental growth: A distributed system can grow or expand more gracefully than a nondistributed one. For example, in adding the
data-storage capacity, it may be easier to add another site to the system without disrupting normal services.

3. **Reliability and availability**: A distributed database system is more reliable than a centralized one because it is constructed from multiple hardware (computers) located at multiple locations (sites). Thus, it is not subject to total failure when one computer breaks down or one geographic location becomes inaccessible. Also, if a datum is replicated, then the datum is always available as long as one of its copies is available. Moreover, each copy doubles as a backup in case the datum is destroyed by some failure.

4. **Efficiency**: Data in a distributed database system can be stored close to their normal point of use, thus reducing both response times and communication costs if most accesses are local. Furthermore, if the pattern of use changes, then data can be dynamically moved or replicated, or existing copies eliminated, to cope with the changing needs.

c. **Difficulties inherent in a distributed database**
The conceptual notions of consistency, concurrency and
transaction still apply in a distributed database. A
distributed database, however, further complicates the issue
of concurrency control. There are a number of reasons for
the complexity (Date, 1983 and Rothnie and Goodman, 1977).

1. The most overwhelming reason is that
communication links are typically rather slow in
comparison with local storage devices such as
disks. Also, communication systems typically
have high access delay. To complicate matters
further, different communication systems have
widely differing characteristics. As pointed out
by Rothnie and Goodman, data rates can vary by as
much as three orders of magnitude and access
delays by as much as six. Hence, design
decisions suitable for one system may be quite
unacceptable for another.

2. In contrast to a centralized database, where the
duplication of data serves as a backup for the
database, copies of data in a distributed
database provide higher performance, because data
are kept near the places of use, in addition to
guarding against failures. The management of
such active copies adds additional complexity to
the control algorithm.

3. Since a key motivation for a distributed database
is the requirement for high database availability, the managing system of a distributed database must be able to guarantee this availability in the presence of failures of system components. Not only does the handling of failures have to be correct, but it also has to be practical and efficient.

There are also other minor considerations such as the structure of the data dictionary. At this point, however, only the first two factors are considered. The remaining factors have been addressed by many researchers but have not been clearly formalized.

d. **Common synchronization techniques**

There are two common techniques which are used to synchronize concurrent transactions in distributed databases: locking and timestamping (Bernstein and Goodman, 1981 and Date, 1983). Although some researchers have used these two techniques as a major basis of classifying concurrency control, they appear, in this research, as a much lower secondary basis for the categorization.

1) **Locking**

This technique is an outgrowth of the usual locks in centralized databases in which different transactions are prevented from simultaneously owning conflicting locks. Usually, in order to guarantee consistency, a transaction must lock its data in a two-phase
manner (Eswaran et al., 1976), i.e. once a transaction surrenders ownership of a lock, it may never obtain additional locks. Locking can result in deadlock in which two or more transactions wait on each other indefinitely. This requires either detection or prevention (avoidance) of deadlock.

2) Timestamping In this technique, each transaction is assigned a unique number called timestamp when it enters the system. Also, this timestamp is usually assigned to the transaction's database actions. DDBMSs are required to process conflicting actions in their timestamp order. Since timestamps usually form a partial ordering, processing of actions will guarantee consistency. There are various ways to generate timestamps. Timestamps may or may not be consecutive numbers. They may or may not constitute a total (linear or simple) ordering. They may be generated distributedly or issued by some central control. Usually, successive timestamps are generated as nondecreasing whole numbers. A timestamp $TS_i$, which is smaller than another timestamp $TS_j$, is said to be earlier than $TS_j$. And the corresponding transaction $T_i$ is said to be older than $T_j$. Conversely, $TS_j$ is later than $TS_i$ and $T_j$ is younger than $T_i$. If the timestamps form a partial ordering and the pair $(T_i, T_j)$ belongs to this ordering, then $T_i$ is said to precede (represented by the symbol "<") $T_j$, and $T_j$ follows $T_i$, in
the partial ordering.

e. Mutual consistency Data in a distributed database can be duplicated. The degree of redundancy may range from none (partitioned or nonredundant) to some (partially redundant) to where each site contains a copy of the whole database (fully redundant). In contrast to a centralized database, copies in a distributed database not only serve as backups in case of failure (in addition to the usual backups in the centralized database local to each site), but also offer the possibility of higher performance since copies can be situated at or near the places of use. The copies, however, pose additional complexity on the consistency requirement. The consistency concerning copies is defined as the mutual consistency, whereas the consistency involving predicates on each copy of the data is defined to be internal consistency (Le Lann, 1978, Milankovic, 1980 and Thomas, 1978). The constraint on copies of a datum is that if all actions in the system were to cease, all copies must eventually take the same value.

B. Essence of Research

1. Problem statement

Because any distributed database is exceedingly complex, quite a number of concurrency control algorithms have been proposed in recent years. No one has yet found a
satisfactory measure for comparing such algorithms. There are at least two reasons why an effective quantitative comparison is elusive.

1. Since distributed database systems are complex and varied, so are concurrency control algorithms. Thus, the essential parameters for quantitative analysis are hard to identify and define. Their complex relationships result in highly complicated and mostly approximated mathematical models.

2. Each group of researchers is acquainted with only a limited number of database systems. As a result, the concurrency control protocol devised in each case is usually based on the actual system at hand. Thus, the details, peculiar to the system, tend to dominate the description of the algorithm, obscuring those features that are common to other algorithms. The result is extreme difficulty in comparing algorithms or proving correctness.

Combined with a lack of a clear understanding of how to approach the problem of concurrency control, each group of researchers has come up with its own solution. Actually, it seems that most of the solutions are ad hoc in that it grows from the database system at hand and the consuming need to
do something about the problem of concurrency control.

As a result of these complications, there seems to be a great number of concurrency control protocols, none of which seems compatible with the others. Since all protocols attempt to preserve consistency, then there must be some meaningful and orderly categorization of algorithms. One aspect of this research is to develop a categorization scheme which can express such a relationship. The second aspect of this research deals with the synthesis of concurrency control algorithms from formal specification of its properties. Papadimitriou (1979) states that there is no obvious neat way to compile syntactic restrictions on concurrent execution into algorithms that achieve them. This research makes some inroads into this area by demonstrating the synthesis of abstract algorithms, both centralized and distributed, from several well known classes of syntactic restrictions.

2. Review of previous categorization schemes

There have been at least seven recent attempts to categorize or compare concurrency control protocols. Bernstein and Goodman (1979) roughly categorize concurrency control protocols based on locking, majority consensus and a special technique for dealing with classes of transaction. Gardarin (1980) introduces an extended petri-net to model algorithms based on the techniques of locking, voting,
preanalysis and distributed control. Wilms (1980) analyzes and compares ten algorithms by a set of measurements such as the number of messages required. The analysis is based on a number of attributes such as the degree of database redundancy and granularity, and logical and physical transmission techniques. An elaborate work by Bernstein and Goodman (1981) employs combinations of locking and timestamping techniques, and their variations, to produce four major categories which break down into 48 possible concurrency control algorithms. Cheng (1981) investigates the performance of models based on a resilient centralized protocol and a distributed protocol involving timestamps. Hsiao and Oszu (1981) survey and classify protocols into locking-based, majority consensus and conflict-analysis approaches. Carey (1983) proposes an abstract algorithmic model to compare storage requirements and CPU overhead among protocols. These protocols are based on locking, timestamping and post-execution validation of serializability.

Some general remarks can be made about these attempts.

1. All attempts at categorization stem from two motivations: either an attempt at a survey of protocols, or an attempt at a quantitative analysis and comparison. For the former, the survey of so many diverse algorithms calls for
some kind of classification. For the latter, in order to systematically analyze and compare algorithms, a survey is also needed to establish the significant parameters to be used in the analysis. In either case, the categorizations are based primarily on two features of the protocols.

- **The apparent features**: These are features which naturally come to mind. The most prominent is the classification of protocols into those with centralized and those with distributed concurrency control algorithms.

- **The superficial features**: These are features which are peculiar to the system but stressed in its description. When considered in-depth, they are often specific implementations of some more general features. Some examples are the majority consensus mechanism, the circulating token, the virtual ring, and the movable (migrating or circulating) centralized control.

2. Some attempts at analysis and comparison, although quantitative in nature, are based on the apparent or superficial features of the protocols. As a result, the comparisons do not
show the relationship among various methods of concurrency control.

3. Most researchers agree that a quantitative comparison is needed in the field of concurrency control. This seems premature since there is still no unifying scheme of categorization on which to base the result of the comparison.

Of all the attempts, only that of Bernstein and Goodman (1981) seems to form a true taxonomy based on fundamental attributes rather than variations in implementation. Although the resulting categorization is impressive, some difficulties still exist since they cite three algorithms that do not fit anywhere in their scheme.

3. Purpose of the research

There are two purposes to this research. The first, and the major one, is the categorization scheme for concurrency control algorithms. The second, and minor, purpose is the synthesis of concurrency control algorithms.

a. Development of categorization scheme  The aim of this research is to devise a categorization scheme for concurrency control algorithms in distributed databases with the following characteristics.

- The categorization is based on a generalized database model. This model is flexible enough to accept variations in database systems.
The categorization is based on the concept of serializability which is widely accepted.

A seemingly natural approach is taken to partition the set of concurrency control algorithms into major categories.

The categorization is generalized and flexible enough to include all protocols studied by the author.

Selection of parameters for the categorization is based on their seemingly relative impact on the problem of concurrency control, rather than on superficial variations in implementation or characteristics of specific databases.

b. Synthesis of concurrency control algorithms

The investigation of various concurrency control algorithms and the formulation of some basis for the categorization scheme leads to an insight on how a concurrency control algorithm can be orderly realized. Such a synthesis of algorithms has the following characteristics.

1. Given mathematical constraints on allowable execution sequences, the synthesis provides an abstract algorithm which only involves concurrency control.

2. A synthesized abstract algorithm may be implemented on any lower level physical
4. **Benefits of the research**

The result of this research is beneficial in two ways. The first is the benefit of the categorization scheme. The second is the benefit from the synthesis of abstract concurrency control algorithms.

a. **Benefit due to the categorization scheme**

The categorization scheme of this research will be beneficial in two ways. First, the categorization will help in the understanding of the concurrency control problem and the more effective use of various solutions. Second, this research may initiate new directions for research in the complicated field of concurrency control.

The categorization itself has several major benefits.

- The categorization stems from a unifying basic concept of serializability. Distracting details of specific characteristics of various database systems are also screened out. As a result, the various algorithms can be placed in the categorization scheme, irrespective of the hardware on which they are implemented.

- The categorization is structurally unifying. This can disclose missing groups of approaches to the problem. The missing algorithms may not be practical, but they may be valuable in the
understanding of the fundamental concepts, and in the fine tuning of other algorithms. Also, the missing algorithms may represent areas of implementation which result from the theoretical concepts that have not been previously explored.

- In the current state of the art, there are, on the one hand, the in-depth theoretical studies of the problem of concurrency control, and, on the other hand, the ad hoc approaches of implementation. This categorization is an attempt at exploring the practical aspects of the theories, and also at extracting the fundamental concepts behind each protocol. Thus, the categorization helps to bridge the gap between theoretical concepts and implementation attempts. This will aid in further understanding of the theories and also on how to go about effectively implementing the theories.

Since distributed concurrency control is in a state of extreme turbulence, the author believes that a solid scheme of categorization will make the situation more manageable. This is because of the elimination of distracting details which are inherent in every protocol. Such details obscure the significant concepts behind protocols, and apparently make it very difficult to initiate any meaningful comparison. Instead, in this research, a natural way to
solve the problem is employed. It is hoped that this natural approach to the problem will result in a meaningful form of categorization, and more important, to show that there is a great need for this kind of undertaking before any major progress can be made on the quantification of the algorithms.

b. Benefit due to synthesis of algorithms Although the synthesis developed in this research is systematic, the author cannot claim that this attempt results in a refined procedure for constructing concurrency control algorithms. However, some potential usefulness of the synthesis is apparent.

1. If, for a given class of execution sequences, constraining mathematical properties can be derived, then the technique can be used to synthesize the abstract algorithm.

2. Mathematical constraints for an algorithm, which are derived from the specified class of resulting execution sequences, tend to concern only concurrency control. The benefits of this are clarity and possible proof of correctness for the algorithm.

3. The technique is demonstrated for three successively contained classes of execution sequences. As a result, the synthesis of a
lesser class can be based on the next larger class.

It is still an open question as to whether the constraining mathematical properties can be identified sufficiently for any given class of execution sequences.

5. Research outline

In the next chapter, Chapter II, twenty algorithms and their variations on concurrency control in a distributed database are reviewed. In Chapter III, the scope, restrictions and some considerations concerning the concurrency control problem are defined and discussed. This framework of concurrency control is represented as the basic distributed database model on which to base any concurrency control algorithm studied in this research. In Chapter IV, previous theoretical work on the concept of serializability is summarized. Various interesting classes of serializable execution sequences are identified and analyzed in order to simplify the family of these classes and isolate a single class, the defining properties of which serve as an underlying basis for the categorization scheme. The scheme of categorization is developed in Chapter V, and each concurrency control algorithm is classified according to the scheme. Chapter VI presents the synthesis of centralized and distributed concurrency control algorithms. This attempt indicates an orderly way to synthesize an abstract
concurrency control algorithm from a set of mathematical constraints. Chapter VII summarizes the result of the research. Benefits and suggested further research are also discussed.
II. SURVEY OF EXISTING CONCURRENCY CONTROL METHODS

In recent years, many concurrency control algorithms for a distributed database have been proposed. This chapter contains a review of twenty such algorithms. This review is important because these methods serve as the basis in selecting useful components for the proposed categorization scheme in this thesis. While it is impossible to do justice to the details of the various algorithms in a review, the survey explicitly mentions those features that were stressed in the associated references. However, the absence of a particular feature in the survey does not imply its absence in the algorithm. The review also provides the reader with an appreciation of the difficulty faced in attempting to form a comprehensive categorization. This is the turbulent situation referred to by Bernstein and Goodman (1981).

A. Description of Methods

To provide a proper perspective, the solutions are presented in chronological order. For each year, however, the listing is in alphabetical order by authors.

The following is the list of the algorithms described in this chapter.

1. Unanimous consensus with full redundancy (Ellis, 1977)
2. Action processing in timestamp order (Badal and Popek, 1978)

3. SDD-1 concurrency control scheme (Bernstein et al., 1978, Bernstein et al., 1980a, Bernstein et al., 1980b and Rothnie et al., 1980)

4. Distributed data-sharing system using tickets (Le Lann, 1978)

5. Distributed deadlock detection (Menasce and Muntz, 1978)

6. Inconsistency avoidance in daisy-chained system (Rosenkrantz et al., 1978)

7. Primary-copy locking scheme (Stonebraker, 1978)

8. Majority consensus with full redundancy (Thomas, 1978)

9. Distributed precedence-relation analysis (Badal, 1979)

10. Centralized control with full redundancy (Garcia-Molina, 1979)

11. Distributed request-queue with full redundancy (Herman and Verjus, 1979)

12. Closely synchronized distributed clock (Kaneko et al., 1979)

13. Reversible unanimous consensus (Rahimi and Franta, 1979)

14. Migrating central control by unanimous consensus
(Seguin et al., 1979)

15. Distributed waiting queue with full redundancy
    (Chou and Liu, 1980)


17. Distributed reservation-list with full redundancy
    (Milenkovic, 1980)

18. Migrating control on virtual ring (Greene, 1981)

19. Deadlock-free resource management with ordered locking of data (Storz, 1982)

20. Multiversion database (Reed, 1983)

1. **Unanimous consensus with full redundancy**

   Ellis (1977) proposes a locking system with a distributed concurrency controller for a fully redundant database. The distributed controller is a set of controllers associated with database sites. Each local controller is physically a process in the DDBMS of that site. A controller, which initiates an update of the database, first broadcasts the attempt at updating to all other controllers and then waits for approval. Upon receiving unanimous consensus which signifies approval of the attempt, the initiating controller updates the local copy of the database. It then broadcasts the update to the other controllers who also perform the update.
The scheme handles mutual consistency explicitly. Preservation of internal consistency is open, depending on the design criteria. Conflicts among transactions, which attempt updates at the same time, are resolved by a priority scheme which assigns a total ordering among transactions. Controller failure is taken care of by a time-out mechanism to avoid indefinite delay. Also, a history file, which keeps track of activities at each site, is introduced to aid in the recovery of controllers.

2. **Action processing in timestamp order**

   In this technique, proposed by Badal and Popek (1978), a set of database actions arriving at a database site is processed in the order of their timestamps. The complexity of the scheme is due to the manner in which sites acquire their actions.

   Basically, when a transaction enters the system the set of sites called read-sites is determined. Each of these sites contains a portion of the transaction's read-set. One of the sites is selected as the preferred site and it acts as the supervisor for all read-sites for this transaction. The preferred site broadcasts to all sites a request to read the read-set. A site will respond to this request if it has produced write actions of some other transactions that must precede any read actions of this transaction. The preferred site then forwards the result of each write action to the
appropriate read-sites of this transaction. After the process is completed, each read-site will have obtained all preceding write actions, which it performs prior to its read action.

The message requirement is claimed to be low since the algorithm allows the result of a write action to be held at its originating site until requested by some preferred site. The result is then forwarded, through the preferred site, to the appropriate read-sites in piggyback fashion on the acknowledgement to the preferred site. The result of a write action may be forwarded in advance of any inquiry if the traffic on the communication network allows. This is to avoid excessive delay in case of low system activity. The problem of reliability is also addressed. In case of some component failure, a transaction can still proceed as long as its data is available. However, the result must be marked as a possible cause of inconsistency. Local logs must be kept in this case to aid in the recovery of the failed site.

3. **SDD-1 concurrency control scheme**

This concurrency control scheme is by far the most complex. It is based on an actual system developed by the Computer Corporation of America. The mechanism of concurrency control in SDD-1 (System of Distributed Database-1) is described by Bernstein et al., 1978,
Bernstein et al., 1980a and Rothnie et al., 1980. The complexity is due to the technique which reduces the volume of communication and also increases the degree of concurrency.

The system uses the distributed timestamp scheme devised by Lamport (1978). The basic idea for concurrency control is to process database actions at a database site in the order of their timestamps. This requires pipelining and guaranteed delivery of messages. Pipelining means messages between two sites are sent and received in timestamp order. In order to reduce the waiting time inherent in such an algorithm, three major techniques are used.

- The employment of the Thomas Write Rule (TWR).
- Possible rejection of read actions.
- Elaborate synchronization protocols.

Employment of TWR: By this scheme, a write action is never rejected, because it is costly. Each datum in the database has an associated timestamp indicating the most recent update of the datum. The TWR rule states that if a write action arrives with the timestamp older than the timestamp of the data to be updated, the action is ignored (Thomas, 1978).

Rejection of a read action: When a read action arrives at a database site, the site processes the action after all older write actions, but before all younger ones.
Thus, if one or more younger write actions have been processed prior to the arrival of the read action, the read action is rejected. This is usually more economical than rejection of write actions, since the transaction is rejected before its computation is started. A special protocol P1 has a feature which allows some flexibility by the initiating site in selecting a timestamp for a transaction. This must be done with care since an old timestamp may result in rejection of the read action, while a young one will cause unnecessary delay because the read action has to wait for possible older write actions.

**Elaborate synchronization protocols:** Transactions are divided into classes determined by their read-set and write-set. Conflicts among classes are represented by a set of undirected graphs, called conflict-graphs, which undergo a complex analysis at the time the system is designed. A set of synchronization protocols, P1 through P4, are used at execution time to enforce the synchronization dictated by the conflict-graphs. The basic idea is that transactions in classes which interfere in a restricted way do not have to be elaborately synchronized. Thus, the four protocols handle various degrees of interference, starting with P1, which takes care of minimal conflict among classes, up to P4, which essentially stops all activities in the system so that abnormal transactions can be processed separately.
Since a transaction has to be synchronized with transactions of other interfering classes, depending on the protocol being used, the system achieves a higher level of concurrency at each site.

4. Distributed data-sharing system using tickets

In this scheme by Le Lann (1978), the system consists of two types of sites: the controllers and the storage processors. The controllers form a virtual ring on the communication network. A message called the token is circulated on this ring. The token carries the system clock(s). Techniques are employed to guarantee the resiliency of the token against failure. The token carries one or more tickets, each of which is a simple counter and acts as a clock for some partition of the database. Each controller, upon receiving the token, will copy the value of one or more tickets as needed. Each ticket used is incremented by one, and the token is sent to the next controller on the ring. The value of the ticket is used as the timestamp for dispatching an action of a transaction. Note that this is different from timestamping the entire transaction. The values of each ticket form a total ordering and are consecutive.

Actions at a database site are processed in the order of their timestamps. The major difference from other schemes is that the timestamps are consecutive.
Two systems of storage processors are proposed: the integrated and the partitioned distributed data-sharing system (DDSS).

**Integrated DDSS**: In this system, a storage processor supervises a complete copy of the database. Thus, there are as many copies as there are processors. This system requires only one ticket in the token. A database action dispatched by a controller is broadcast to all processors. Each processor executes actions in consecutive order of the ticket values. The processor simply waits for any missing ticket value.

**Partitioned DDSS**: Each storage processor supervises only a portion, or a partition, of the database. The partition may or may not overlap with partitions of other processors. In this system, the token contains a number of tickets, one for each partition. A controller selects the appropriate tickets when the token arrives. The selection depends on which partitions are to be affected by the transaction. As a ticket only deals out values for transactions concerning the ticket's partition, the actions arriving at that partition will carry consecutive ticket values. Processing of actions at a storage processor is thus a straightforward process.
5. **Distributed locking**

Concurrency control solutions mentioned so far rely on the ordering of transactions through timestamps. In this work by Menasce and Muntz (1978), a transaction is allowed to proceed without any prior ordering with respect to other transactions. A locking technique is used to ensure consistency. Inconsistency results in deadlock, where the locking requirements of two or more transactions form a cycle in which each transaction has to wait for the other(s) to release some locks. This scheme favors distributed deadlock detection which resolves deadlock after it has occurred. Two approaches to distributed detection are proposed.

- Hierarchical deadlock detection.
- Distributed deadlock detection.

Both methods rely on a graphic model called transaction wait for (TWF) graph. Transactions are represented as vertices, while a wait-for relationship is a directed arc from the transaction being blocked to the transaction causing the blockage. A cycle in the TWF graph indicates the existence of a deadlock. Deadlock resolution involves the selection of one or more transactions to be preempted in order for the cycle to be broken.

**Hierarchical detection:** This method of deadlock detection employs a set of controllers arranged in a tree
Each leaf-controller is associated with a database site. Each controller tries to locate and resolve any cycle using pieces of TWF graph produced by its descendant controllers in the hierarchy. The remaining TWF graph is simplified and forwarded up the hierarchy to the controller's ancestor. Thus, any waiting between two nonoverlapping subhierarchies will be handled by the controller which is their common ancestor. In the worst case, a cycle can only be detected and resolved by the root controller. Nevertheless, information is constantly being condensed as it goes up the hierarchy. Thus, the message requirement can be reduced. Also, there is a possibility that cycles can be detected at a lower level of the hierarchy. In this way, a reduction in delay can be substantial.

**Distributed detection:** In this method, each database site generates pieces of the TWF graph based on its knowledge of various transactions. This piece of TWF graph is forwarded to sites which control the transactions involved in the graph. Each controller site tries to detect and resolve cycles based on the received pieces of the TWF graph. This method offers a certain degree of redundancy because of the duplication of each piece of graph. On the other hand, the duplication places more load on the network.
6. **Inconsistency avoidance in daisy-chained system**

In this system by Rosenkrantz et al. (1978), a transaction is executed by a set of processes which are initiated in a daisy-chain manner at various sites. The initiation of processes depends on the need for remote data. When a process $P_i$, at database site $S_i$, needs data at some remote site $S_j$, it initiates a process $P_j$ at site $S_j$. Then, $P_j$ will be active while $P_i$ becomes inactive. Thus, a transaction progresses from site to site, leaving a trail of its progression in a daisy-chained fashion. The only active part is the process invoked at the head of the chain. A process is continued after the process receives a termination message from its descendant. Although no explicit locking requirement is mentioned, the access of data by a process can cause processes, in support of other transactions, to wait. Thus, there is the possibility of deadlock.

The scheme employs deadlock avoidance. Two approaches were proposed: the wait-die and the wound-wait protocols. For both protocols, a numbering technique assigns a unique number to each transaction. A transaction with the smaller number is said to be older, and one with the larger number is said to be younger.

**Wait-die protocol:** In this protocol, if a transaction $T_j$ arrives at a site and conflicts with another
transaction $T_i$ already at that site, then $T_j$ waits if it is older than $T_i$, otherwise $T_j$ dies. When a transaction dies, a message is broadcast to all of its processes to effect the rollback of the transaction. The system is guaranteed to avoid deadlock because the youngest transaction in a cycle of waits will eventually be rolled back and the cycle will be broken.

**Wound-wait protocol:** In this case, if $T_i$ conflicts with $T_j$ and $T_i$ is older than $T_j$, then $T_i$ wounds $T_j$, but if $T_i$ is younger, then $T_i$ waits. When a transaction is wounded, it broadcasts a message informing its processes of the wounding. If the message gets to a site that contains the active process of the transaction and the process has not initiated a termination, then the transaction is aborted. The system is free of deadlock because the youngest transaction in a cycle will eventually be wounded by some other transaction.

7. **Primary-copy locking scheme**

This method, proposed by Stonebraker (1978), is intended to support a partially redundant database. For duplicated data, a primary copy is designated. To preserve internal and mutual consistency, a transaction must obtain locks on all of its data at the primary site(s) before any update. Each site has a list to help find the primary copies. Unanimous consensus of all operational sites is
needed to establish a new primary copy.

This method of concurrency control recognizes two levels of data consistency. The first level is where a transaction is read-only and the user is not concerned with consistency. At this level, a transaction accesses local copies of the required data. Any datum absent from the local database is accessed through the primary copy. The second level of consistency is where the user requires normal consistency constraints for a read-only transaction. In this case, every access of a transaction is done through the primary copy. There is no separation of consistency level if the transaction updates the database. Each access of such a transaction must always pass through a primary copy.

Since deadlock can occur, the method employs a centralized deadlock detection scheme. A database site is unanimously chosen to act as the detector. Such a site is called "SNOOP". When a certain site detects that one transaction is waiting for another transaction, the site sends the information to SNOOP. SNOOP gathers such information into a system-wide wait-for graph to detect any deadlock. In order not to overburden SNOOP and the communication network, each site also tries to locally detect and resolve deadlock if possible.

The proposed scheme also addresses the problem of
failures. An up-list is maintained at each site. The system reconfigures itself if a site fails. The reconfiguration employs unanimous consensus from all sites that are operational. In the case where a failure causes a partitioning of the network into two or more disjoint portions, a partition must contain a majority of copies of a datum in order to legally represent that datum. The copies, which are quiescent, in other partitions can be brought up to date when the partitions are joined.

8. Majority consensus with full redundancy

This scheme by Thomas (1978) uses a majority consensus in committing a transaction. The scheme employs a minor variation of the distributed clock proposed by Lamport (1978). Instead of using timestamps in messages to keep clocks at various database sites running close to each other, the clocks are synchronized through timestamps affixed to data in the database. The database is assumed to consist of a collection of named elements, each of which has a value and a timestamp. The timestamp is that of the latest transaction which updates the value of the element. A transaction's timestamp consists of two parts. The first part is the identifier of the originating site of the transaction. The other part is obtained from the site's counter or the latest (youngest) timestamp of the transaction's read-set, whichever is younger.
The basic idea of this concurrency control method is that if a transaction's read-set is obsolete (overwritten in part or whole by the action of some younger transaction) at some site at the time of the transaction's attempt to write, then the transaction is aborted and has to be tried again.

Since the system is based on full redundancy, a transaction acquires its read-set locally at the originating site. The transaction operates on the data and produces the result which is the write-set. The site then forwards the write-set, together with the timestamps of the read-set, to all the other sites. Two forwarding techniques are suggested: broadcasting and daisy-chaining. Majority consensus is required in order to commit a transaction. In order to handle complications due to transaction conflicts, a site is given four alternatives in casting its votes. An ACCEPT is voiced if no data in the transaction's read-set is obsolete and if the transaction does not conflict with any known transaction. If one or more data is obsolete, then a REJECT vote is cast. A PASS vote is cast if the transaction conflicts with some other known transaction with higher priority. This type of vote helps prevent deadlock if there is a tie. A site defers its vote if the transaction conflicts with some transaction with lower priority. Deferred transactions are reconsidered each time there is a new development at the site. Such a development may be an
acceptance or a rejection of some transaction.

Another major feature introduced is the Update Application Rule, or Thomas Write Rule (TWR). The rule states that if an update is to be applied to a datum, the timestamp of the update is compared with that of the datum. If the former is younger, then the update is performed. Otherwise, it is omitted since the update is obsolete. This rule is intended to take care of the situation where updates arrive at a site out of their timestamp order.

9. Distributed precedence-relation analysis

In this method by Badal (1979), inconsistency is detected using a distributed analyzer. A precedence relation between two transactions is the actual order of accessings of the transactions of a common datum at the same site. Inconsistency arises if the precedence relation among a group of transactions forms a cycle.

This type of solution does not require timestamps, although a priority scheme is needed to break the cycle. Accessings of data at a site establish portions of precedence relation. By exchanging and analyzing such portions among sites, cycles can be detected and resolved. When a transaction is completed but is not yet committed, the various portions of the precedence relation concerning the transaction are analyzed.
10. **Centralized control with full redundancy**

In the schemes by Garcia-Molina (1979), a centralized concurrency controller is employed. Four different schemes are proposed to speed up the process and reduce the bottleneck of a central controller.

- Complete centralization algorithm (CCA)
- Centralized locking algorithm (CLA)
- Centralized locking algorithm with "wait for" lists (WCLA)
- Centralized locking algorithm with "hole" lists (MCLA)

**CCA:** In this algorithm, a database site forwards the transaction to a central site. The central site then reads its local copy of the database, performs the calculation to produce a result, and then broadcasts the result as update to all sites. The update is also given a sequence number. Each site applies updates to its copy of the database in the order of sequence numbers.

**CLA:** In the CCA, the load at the central site is high, potentially causing a bottleneck. The CLA delegates the reading and processing for each transaction to the originating site. A site only requests locks for the transaction through the central site. If the transaction conflicts with some other transaction, it is queued. Lock requests are performed on data in some predefined order to
prevent deadlock. After all locks are granted, the originating site reads the data locally and performs the calculation. It then broadcasts the update to all sites. A sequence number, issued by the central site, also accompanies the update.

**WCLA**: In the CLA, a site which has been granted the locks for its transaction T still has to wait until all preceding transactions' updates have been received and applied to the local database. This may cause unnecessary delay. In WCLA, the central site maintains a table of the last transaction accessing each datum. When a site is granted its locks for a transaction T, the central site constructs a wait-for list for T containing all conflicting transactions which precede T. Thus, the originating site of T has to wait only for the completion of those transactions in the wait-for list.

**MCLA**: Construction of the lists in the WCLA may require an excessive amount of memory. Also, the checking required before processing a transaction may be lengthy. In MCLA a hole list is constructed for a transaction T. This list identifies those transactions that do not conflict with T but still have not released their locks. Thus, to process T, a site does not have to wait for transactions in the hole list. Still, the site has to wait for updates of other transactions which have released their locks. Such
transactions may or may not conflict with T. Although the updates of such transactions may be delayed in the broadcasting process, the waiting is not unreasonable since the transactions have been completed.

11. **Distributed request queue with full redundancy**

   This method by Herman and Verjus (1979) is based on a system of distributed request queues. Events or happenings are totally ordered by a system of distributed timestamps, the nature of which is similar to that described by Lamport (1978). The database is fully redundant and each copy of the database is considered an indivisible entity.

   In a system of distributed request queues, a database action, upon arriving at a site, is placed in the site's request queue in timestamp order relative to other actions already in the queue. A site always selects the action with the earliest (oldest) timestamp in the queue for processing. In order to guarantee that no transaction with an intermediate timestamp is missing, two constraints are employed.

   1. Actions sent and received between any two sites are in timestamp order, i.e. they are pipelined. Also, communication channels are assumed to be reliable.

   2. An action is deemed oldest in a request queue if it has the earliest timestamp in the queue and
there exists, in the queue, at least one action from each of the other sites.

In this method, transactions are grouped into three classes: READ, WRITE and UPDATE. A READ only reads data, hence it consists of a read action only. Since the database is fully redundant, a READ can obtain its data from the local database copy. Nevertheless, the READ still has to compete with other local and nonlocal actions through the local request queue. A WRITE only writes data, and hence consists of a write action only. It has to be broadcast to all sites and competes with other actions at each site (including the local site) through the request queue. An UPDATE consists of a READ followed by a WRITE, and the handling of each action is the same as that of a READ and a WRITE transaction, respectively. If an UPDATE transaction has more than one READ or WRITE step, then brackets have to be employed to group such READ (or WRITE) steps into an indivisible step.

12. **Closely synchronized distributed clock**

In this work by Kaneko et al. (1979), a logical clock synchronization method for fully redundant database is proposed. The method requires close synchronization of clock values at different sites.

A database site maintains its local clock as follows. When the site advances its clock from time $k-1$ to $k$, it
broadcasts a message with timestamp $k$ to every other site. Also, a site changes its clock from $k-1$ to $k$ only when it has received messages with timestamp $k-1$ from all the other sites. In such a system, the clock of a site differs at most by one clock-tick from any other site. Furthermore, at time $k+2$, a site will have received all messages with timestamp $k$. Thus, the basic idea for concurrency control is for a site, at any time $k+2$, to examine a request for update $U$ with timestamp $k$. If the read-set of $U$ is obsolete or if $U$ conflicts with other updates with the same timestamp but with higher priority, then $U$ is rejected.

13. Reversible unanimous consensus

This algorithm (Rahimi and Franta, 1979) is different from that of Ellis' (1977) in one major respect. In this scheme, a vote to accept a transaction by some site can be reversed if the voting process for the transaction has not been completed. When a database site receives a request for an update, it votes to accept the update if there is no pending conflicting request at the moment. In order to resolve conflicts among transactions, a timestamp scheme similar to that of Lamport (1978) is used although no attempt is made to keep the values of the timestamps close together. A request is rejected if it conflicts with another pending high-priority request. If the request conflicts with a pending low-priority request $R$, the site
inquires of the originating site of R whether or not it can reverse its vote on R. If the originating site of R has already received unanimous consensus on R, then the vote cannot be reversed. Otherwise, R is aborted and the inquiring site is granted the reversal.

The algorithm also allows for higher degree of concurrency by partitioning the database into independent sections and maintaining a queue for each partition. The queue is used in case the application of updates lags behind the voting process.

14. Migrating central control by unanimous consensus

This scheme (Seguin et al., 1979) is applicable in a fully redundant database. When a site wishes to perform an update, it tries to acquire the status of a central controller, called the administrator. It acquires the status by asking permission from the current administrator. Contention at this stage is resolved simply by granting the first request and rejecting all others. The rejected requests, however, are queued and the queue is forwarded to the new administrator. The new administrator then broadcasts its newly acquired status and asks for acceptance from all sites. After all acknowledgements are received, the administrator broadcasts the update. Majority consensus is required at this stage to effect the installation of the update, but the consensus is only a fail-safe technique for
the second phase of the two-phase commit protocol and not really a part of the concurrency control technique. Any request for update that arises during this stage is queued.

There are also mechanisms to handle abnormal conditions such as the inclusion of a new site and the failure of the administrator. In each case, some form of consensus is required so as to guarantee that all sites have the same view. The majority consensus is also required in case of partitioning of the network due to failure.

15. **Distributed waiting queue with full redundancy**

This method of concurrency control (Chou and Liu, 1980) is intended for a fully redundant database. Each copy is considered an indivisible entity and, as a consequence, a single queue of actions (equivalent to the request queue by Herman and Verjus, 1979) is maintained in timestamp order at each site. As soon as an action in the queue is determined to be the oldest possible action, it is removed from the queue and forwarded to the controller of the database local to that site. This allows for some overlap and the controller can exercise centralized control to process the transactions concurrently. The solution suggests a simple method of locking for local concurrency where a transaction is granted either all locks or none in order to avoid deadlock. This scheme assumes that delay involved in ascertaining which action has the oldest timestamp is not
excessive so that the drawback of the single waiting queue is not serious.

There are a number of suggestions to handle abnormal conditions. The system uses unanimous consensus of active sites to determine if a site has failed. If a communication failure causes a partitioning of the network, a partition can continue to function if it has a majority of the sites.

16. Centralized serializability controller

This scheme (Dewitt and Wilkinson, 1980) is intended for a local broadcasting network with a partitioned database and no redundancy. The nature of the network makes it efficient for a single site, called the concurrency control node, to eavesdrop on the traffic among sites. Thus, a transaction is processed without any constraint up until it is ready to be committed. The control node, which collects and maintains the interactions among transactions, decides whether a transaction is to be committed or not.

Consistency preservation is based on serializability. In this approach, suppose an action $A_1$ of a transaction $T_1$ reads a datum $X$. If another action $A_2$ of a transaction $T_2$ writes $X$ later on, then $T_1$ must precede $T_2$ in the serialization of transactions. Since $T_1$ may still be active, there is the possibility that the later actions of $T_1$ will conflict and follow actions of $T_2$ causing a nonserializability situation. There are three approaches in
avoiding such a situation.

- Starvation method
- Non-starvation method
- Restriction-list method

The starvation method requires $T_2$ to be committed after $T_1$. In the meantime, there may be any number of transactions reading the datum $X$, thereby causing $T_2$ to be delayed indefinitely. In the nonstarvation method, if $T_2$ is completed and is waiting to be committed, the control node will flag $T_2$ as golden. Any other transaction accessing $X$ after this point is rejected. Thus, $T_2$ has to only wait for $T_1$. Still, this method may cause a lot of rejections. The third approach, the restriction-list method, allows $T_2$ to be committed without waiting for $T_1$. In this case, a restriction is imposed on $T_1$. If $T_1$ accesses $X$ again, it has to be aborted so as not to create a cycle in the precedence relation between itself and $T_2$. Thus, no waiting is necessary in this method.

17. Distributed reservation list with full redundancy

The basic idea of this method by Milenkovic (1980) is similar to that of Chou and Liu (1980). It is designed for a fully redundant database. But instead of considering each copy of the database as indivisible, this scheme assigns at each site a separate waiting queue (reservation-list) for each datum. There also are two approaches to the scheme:
the pessimistic (P) and the optimistic (O) protocol. Both protocols employ the timestamp scheme suggested by Lamport (1978). Also, pipelining and reliable transmission are assumed.

**P protocol**: In this approach, the read-set and write-set of a transaction is entered in the appropriate reservation-lists at the initiating site. The entries are inserted in timestamp order. The site broadcasts the identity of the write-set to all other sites and waits until either each site has acknowledged the broadcast, or a message with younger timestamp has been received from each site. The transaction can then be processed and the value of the write-set broadcast for installation in all copies. This protocol requires the construction of read-sets and write-sets from all variables declared in the transaction, regardless of whether or not they are actually needed at run-time. This requirement may result in higher level of conflict among transactions.

**O protocol**: Unlike the P protocol, this protocol requires the construction of the read-sets and write-sets based on actual needs. The O protocol is designed for systems with little conflict. A transaction is executed by the initiating site using the local database copy. The site then asks all other sites if the read-set of the transaction is obsolete or not. If it is, the transaction is aborted,
else the write-set is broadcast. This protocol is similar to that by Thomas (1978) except for the use of unanimous consensus in determining the success of a transaction.

18. **Migrating control on virtual ring**

The database in this scheme (Greene, 1981) is fully redundant. Database sites form a virtual ring with a fixed configuration. Thus, the identities of the predecessor and the successor of a site are known. A site, after receiving permission and relevant information from its predecessor, processes local transactions and passes on the privilege to its successor after a specific time quantum. The information passed from site to site is the accumulated updates and locks performed on each site. When a site takes control, it appends any updates generated locally to the received update-list and applies the update-list to the local database copy. The site then performs any locking required by read actions of local transactions. Such locks must not conflict with the ones shown in the received locking-list. When its time quantum has expired, the site forwards the update-list and the locking-list to its successor.

Each entry in the update-list has a counter associated with it. When a site applies the entry to its local database copy, the counter is incremented by one. An entry is deleted from the list when the counter registers the
value equal to the number of sites on the ring.

Deadlock is prevented by requiring a transaction to obtain all of its data before processing. If there are a lot of conflicts, obtaining all locks of a transaction may span several control quanta of a site.

19. **Deadlock-free resource management with ordered locking of data**

Transactions in this scheme (Storz, 1982) are grouped into r-transactions, each of which contains only read actions, and r/w-transactions, each of which contains both read and write actions. r/w-transactions use a two-phase-lock technique (Eswaran et al., 1975) to access data, while r-transactions do not lock data. There are two concurrency control algorithms, one for each type of transaction. For an r-transaction, its conflicts with other r/w-transactions are collected and analyzed. If the result can violate the consistency of the database, then the r-transaction is restarted. The control algorithm for r/w-transactions is more complex because the processing of an r/w-transaction is divided into four phases of activity: read, lock-for-write, ready-to-write and write phases. Some of these phases cannot cause inconsistency while other can cause inconsistency in a local database. System-wide inconsistency is prevent by an ordered locking of data with respect to locations of sites. To prevent local
inconsistency, some transactions may have to be restarted. Selection of the r/w-transaction to be restarted depends on the phase of activity of the transaction and the interactions with other r/w-transactions.

20. Multiversion database

In this scheme, proposed by Reed (1983), the system retains recent versions (states) of each datum. These versions are ordered by a temporal-vector called pseudo-time (pt) private to the system. Versions express the history of a datum's life in the form of database actions performed on the datum. A version occupies a continuous interval of the pt stream. Such an interval is termed a pseudo-time span (pt span). The pt span of a version extends from the pt of the version's creation (called definition) by some write action, to the time of the version's latest access (called look-up) by some read action. The pt spans of various versions of a datum do not overlap. Furthermore, a version can be defined only once.

A read action may try to look up a datum either at a designated pt within the pt span of some version or at a pt where no version exists. In the former case, the version to be read is obvious. In the latter case, the pt span of the version, immediately prior to the designated pt, is then extended to include these designated pt. In this system, no read action is ever rejected.
A successful write action must necessarily define a new version. If the write action's pt does not fall within the pt span of any existing version, then a new version can be defined (created) at that pt. If, on the other hand, the write action's pt falls within some version's pt span, then the write action is attempting a redefinition of that version and hence has to be rejected. A redefinition of a version cannot be allowed since it will invalidate at least one earlier (in real time) look-up of that version.

B. Summary

The author does not claim this list of concurrency control algorithms to be complete. It does include all algorithms cited in other surveys known to this author. Also it represents the seemingly diversified approaches to the problem of distributed concurrency control. It will be seen in Chapter VI that the categorization scheme being proposed will include all of these algorithms.
III. FRAMEWORK ON PROBLEM OF CONCURRENCY CONTROL

This chapter establishes the scope of the environment in which a concurrency control algorithm has to operate. This limitation in scope governs the rest of this research. This is necessary since the field of distributed databases is complex and various aspects of the field relate to each other in many ways. Such relationships, if not bounded, can lead to an overly complicated investigation of any one aspect. Section A of this chapter limits the scope of the causes of inconsistency to one that is effectively manageable and formally understandable to some degree. Section B concerns the underlying logical components and environment which together support a concurrency control algorithm. These limitations in scope are synthesized into a model of the distributed database presented in Section C. This model can serve any concurrency control algorithm studied in this research. Also, the model establishes a flexible and simple framework which is common to all algorithms, and thus aids in the task of developing a categorization scheme.

A. Causes of Inconsistency

One of the major requirements of a distributed database is to provide valid data to the user. This means that a user must see the database in a consistent state in order to
manipulate the data. A database is said to be in a consistent state if the predetermined predicates imposed on the data are not violated (Eswaran et al., 1976). These predicates are termed the **consistency constraints**. Usually, these predicates comprise a super structure, defined implicitly or explicitly between the users and the system, on the database. For example, any value added to a datum must be reflected in the increasing value of that datum. This means that if two database actions each increments a datum by 1, at the end of the sequence the value of the datum must be greater than its original value by 2. A predicate may be more complicated and involve a number of data. As an example, consider the population of a country. Barring births, deaths, immigrants and emigrants, the total population must remain constant. In this case, it may be impossible to observe the value of the supposed constant at any given instant. Nevertheless, the algorithm must make sure that a person moving (disappearing) from one place must eventually appear at another.

If a distributed database is considered a computational machine, there are three sources of inconsistency: erroneous inputs, concurrent execution, and the hardware.
1. **Cause of inconsistency due to erroneous inputs**

To preserve the consistency of data, not only must a transaction see a consistent database, it must also leave the database in a consistent state after it is through with the interactions. That is, if a transaction is executed alone in the system, it must preserve the consistency of the database. The responsibility of enforcing this requirement can either fall upon the application programmers, or upon the operating system (Rothnie and Goodman, 1977). The latter choice is more desirable, since the checking is done automatically. Still, the complexity of the consistency predicates may impose a limitation on how far this automated checking can go. Thus, the screening of undesirable transactions is highly application oriented. Hence in this research, every transaction, if executed alone, is assumed to preserve the consistency of the database.

2. **Cause of inconsistency due to concurrent execution**

A requirement of this computational machine is that transactions are allowed to proceed concurrently. The problem of concurrent accesses to common resources is well-understood in conventional operating systems (Deitel, 1983). A distributed database is also a shared resource but, as pointed out in Chapter I, it is far more complicated. If there is no control, the freely executed transactions can eventually lead to an inconsistent database. It is this
source of inconsistency due to concurrent execution that is considered in this research.

3. Cause of inconsistency due to hardware

Like any kind of machinery, the distributed database is subject to failures of its components, whether software or hardware. For the former type of failures, if the algorithm is theoretically sound and correctly implemented, its operational failures can be excluded. The hardware, on the other hand, is prone to failure even after being fully tested. The consistency of the database is thus affected by such failures and, because the variability of component failures is considerable, no comprehensive framework to cope with them has yet been devised. Hence, the problem of hardware reliability is not considered in this research.

In summary, the preservation of consistency requires some kind of restriction on the concurrent execution of transactions. The concurrency control deals with the machination of such restrictions, the nature of which is rather involved, and is taken up in detail in the next chapter.

B. Environment for the Control of Concurrency

The environment in which the preservation of consistency must operate is in effect the distributed databases themselves. In this research, only those aspects
of the database system which influence the concurrency control algorithm are considered. Such aspects are

- organization of data,
- data dictionary,
- network architecture, and
- transaction model.

1. **Organization of data**

In a distributed database, there are two levels of the organization of data: the local and the global organization.

In the local organization, the data contained in a node, called the database site, of the distributed database can be organized in various ways. Nevertheless, in this research, it is assumed that the effect of the organization is transparent to a concurrency control algorithm. The local database controller acts as the interface between the site and the network so that a site is considered to contain a collection of addressable data. The procedure by which each datum is accessed is of no concern to the global database controller. Thus, the local organization is ignored in this research.

There are three main types of global organization: the relational, the hierarchical and the network models. However, the details of such a complex organization add nothing conceptually to the problem of concurrency control and serve to hide the real underlying issue. Hence, it is
assumed that the database is a collection of data
distributed over various sites in the network. Also, each
datum is equally addressable through a table, called the
data dictionary, which is maintained by the system.

In a distributed database, duplication of data is used
for the purposes of reliability and performance. This
degree of redundancy in the organization of data may range
anywhere from one extreme of a fully redundant database to
the other extreme of a partitioned nonredundant database.
This factor is not of primary concern in the categorization
scheme developed in this research. Its presence in a
particular concurrency control algorithm only indicates that
the algorithm is a specialized member of a more generalized
category. This research considers each duplicate as a
separate, distinct datum. Although the syntax of a
transaction does not concern itself with the duplication,
the system can consult the data dictionary for the location
of the duplicates. The original syntactic information of
the transaction can then be extended to include the presence
of copies. For example, if $X_1$ and $X_2$ are copies of a datum
$X$, then an action $\text{write}(X)$ is extended to mean $\text{write}(X_1, X_2)$. 
This generalization implies that mutual consistency can be
translated into internal consistency. In the example, a
usual internal consistency constraint, induced by the
extension, is $X_1 = X_2$. 
2. **Data dictionary**

To access the data in a distributed database, a relation called the *data dictionary*, *directory*, or *schema* is used. In its simplest form, the dictionary is a table which relates the identity of a datum to its location in the system. The structure of the dictionary depends on many factors. The dictionary may or may not be redundant. It may be centralized or distributed. Rothnie and Goodman have suggested three factors which affect the dictionary (Rothnie and Goodman, 1977). High frequency of accesses and the requirement of reliability on the dictionary are two of the factors which encourage a distributed and redundant organization. On the other hand, the third factor which is the high frequency of updates or changes, discourages distribution of the dictionary.

In a sense, the data dictionary is a database. If it is distributed, any change will require the concurrency control similar to that used for the usual distributed database. In the SDD-1 system, the dictionary is considered a part of the database, and as such it is treated as normal data (Rothnie and Goodman, 1980). Although this approach allows a flexible system for the dictionary, the scheme implies a hierarchy of dependencies between one group of data (the dictionary) over the other group (the normal data). In such a case, some hierarchical locking structure
may have to be employed to lock out any access to the data under the part of the dictionary being modified. Locking strategies proposed by Gray et al. (1975) may be suitable for this situation.

Although the dictionary affects the complexity and performance of the concurrency control protocol (Date, 1983), it does not have significant impact on the nature of the problem. Thus, in order to simplify the intended categorization, it is assumed in this research that the database controller maintains a dictionary, a copy of which is kept at each database site. In other words, the dictionary is in itself a fully redundant distributed database. Also, it is assumed that there will be infrequent changes in the dictionary. A change, or update, is reflected in the dictionary through an agreement of all operational sites (a log is kept in this case to bring a crash site up to date). This agreement is carried out by effectively stopping all transactions in the system. Thus, an update on the dictionary is considered to be in a special class of transaction. Such a transaction is assumed to interfere with every other transaction through the dictionary. Handling of this type of transaction requires some special protocol such as that employed in the SDD-1 system (Bernstein et al., 1980a).
3. Physical network architecture

The distributed database is a collection of database sites which are interconnected by some form of communication network. At this point, the nature of the physical network is not important. The only requirement is that the network is reliable. However, additional requirements will be introduced later on to suit specific algorithms.

4. Transaction model

In this research, the model of the simple dual-stepped transaction is adopted. The justification for the choice is discussed here in the context of various types of transactions.

Transactions can be divided roughly into three groups. A simple or mono-stepped transaction has only one interaction with the database and that interaction is considered indivisible. An example of such a transaction is the typical semaphore operation in conventional operating systems. A transaction is called simple dual-stepped if it consists of two interactions with the database. The first interaction is the read-phase, where all elements of the required read-set are obtained from the database. The second interaction is the write-phase, where all of the results, or write-set, are deposited onto the database. The computation-phase, which obtains the results, is not essential as it is transparent to the concurrency controller.
which uses only the syntactic information of the transaction. This type of transaction is used by Bernstein et al. (1979) as the basis for analysis of concurrency control. The last type of transaction is the complex or multi-stepped transaction which contains more than one interaction with the database. The simple dual-stepped transaction is a special case of the complex transaction as is the mono-stepped transaction. But in a simple dual-stepped transaction, there must be a read-phase followed by a write-phase. In the case where a complex transaction has only two actions, no restriction is placed on the type of either action. For example, the transaction may have a read-phase followed by another read-phase or have a write-phase followed by a read-phase.

In general, a database should be able to accommodate any type of transaction. However, because the major portion of this research assumes simple dual-stepped transactions, conversion of a complex transaction into a simple dual-stepped transaction is discussed in the following subsections.

a. **Collapsing the complex transaction** A transaction can be expressed in simple dual-stepped form by forcing extra dependencies among data onto the transaction. Consider the following transaction with more than two database actions.
\[ T_1 : \text{begin} \]
\[
\text{read } X; \ Z=X+1; \text{ write } Z; \\
\text{read } Y; \ W=Y-1; \text{ write } W \\
\text{end.}
\]

From the data dependency standpoint, \( Z \) depends on the value of \( X \) while \( W \) depends on that of \( Y \). At first glance, one may be tempted to split \( T_1 \) into two independent transactions, one of which involves only \( X \) and \( Z \) while the other involves only \( Y \) and \( W \). This cannot be done because there may be a consistency constraint such that the sum of \( X \) and \( Y \) is always equal to the sum of \( Z \) and \( W \). Therefore, \( T_1 \) must be executed in its entirety and considered indivisible by the user. The dependencies among the four data lie beyond the syntax of the transaction. Since the current state of the art has not been able to provide effective means of semantic analysis of the transaction's contents, let alone the hidden consistency constraints, the syntax of a transaction is thus the only effective information that is available for analysis. It is safe to assert extraneous or redundant dependencies inside a transaction and, as a result, the above transaction can be expressed as
\[ T'_1 : \text{begin} \]
\[ \text{read } (X,Y); \]
\[ Z=X+1; \; W=Y-1; \]
\[ \text{write } (Z,W) \]
\[ \text{end.} \]

The result is a simple dual-stepped transaction, with the implication that both \( Z \) and \( W \) depend on the values of \( X \) and \( Y \). It should be noted that the above argument holds true even though \( Z \) is written before the reading of \( Y \) in \( T'_1 \). The extraneous dependencies do not affect the semantics of the transaction in any way.

Similarly, mono-stepped and complex dual-stepped transactions can be expressed as simple dual-stepped form. For example, the transactions

\[ T_2 : \text{begin} \]
\[ X=2; \text{write } X \]
\[ \text{end.} \]
\[ T_3 : \text{begin} \]
\[ X=2; \text{write } X; \]
\[ Y=3; \text{write } Y \]
\[ \text{end.} \]

can be expressed as
\[ T_2' : \text{begin} \]
\[ \quad \text{read NULL;} \]
\[ \quad X=2; \]
\[ \quad \text{write } X \]
\[ \quad \text{end.} \]

\[ T_3' : \text{begin} \]
\[ \quad \text{read NULL;} \]
\[ \quad X=2; \ Y=3; \]
\[ \quad \text{write (}X,Y)\]
\[ \quad \text{end.} \]

where NULL is an empty set of data.

b. Multi-traced transaction Consider the transaction

\[ T_4 : \text{begin} \]
\[ \quad \text{read } X; \]
\[ \quad \text{if (}X=0\text{) then begin} \]
\[ \quad \quad \text{read } Y; \ Z=Y; \ \text{write } Z \]
\[ \quad \text{end} \]
\[ \quad \text{else begin} \]
\[ \quad \quad \text{read } U; \ V=U; \ \text{write } V \]
\[ \quad \text{end} \]
\[ \quad \text{end} \]

The five data in \( T_4 \) comprise the static execution data set (static EDS) of \( T_4 \). From the semantics of \( T_4 \), the use of either \( Y \) and \( Z \), or of \( U \) and \( V \) depends on the value of \( X \), which is not known until \( X \) has been read. Thus, the static EDS of a transaction is defined as the set of all data which appear syntactically in the database actions of the transaction. On the other hand, the dynamic EDS of a
transaction is the set of data which is actually used during the execution of the transaction. Thus, the static EDS of \( T_4 \) is the set \((X,Y,Z,U,V)\), while the dynamic one is either \((X,Y,Z)\) or \((X,U,V)\).

A transaction which exhibits choices of its dynamic EDS is defined as the multi-traced transaction. The simple dual-stepped representation of such a transaction can show only the static EDS. For example, the above transaction \( T_4 \) can be represented as

\[
T_4' : \text{begin}
\text{read (X,Y,U);} \\
\text{computation;} \\
\text{write (Z,V)} \\
\text{end.}
\]

A large difference between the sizes of the static and the dynamic EDS may result in performance problem. A transaction may actually need only a handful of data, but which handful is needed depends on the outcome of some intermediate results. The possible candidates for such data may involve a good portion of the database. In this case, the requirement of static EDS will result in heavy interference among transactions. In the system where dynamic EDS is employed, selective reading and writing can be done to lessen the conflicts. In this case, the syntax
of the simple dual-stepped model serves as a guide, and not an exact rule for the transaction to follow. The choice of either type of EDS will depend on the approach to the concurrency control.

c. Intra-dependent transaction An intra-dependent transaction is a complex transaction which alters the database and later explicitly reads the changed data. For example, consider the transaction

\[
T_5 : \begin{align*}
\text{read } X &; \text{ write } Y; \\
\text{read } Z &; \text{ write } W; \\
\text{read } Y &; \text{ write } U \\
\text{end.}
\end{align*}
\]

The datum Y acts as a temporary working area. The problem here is that its value is also reflected in the database state. Thus, the equivalent simple dual-stepped transaction must contain Y in its write-set. However, the following transaction is not a good representative of $T_5$.

\[
T_5' : \begin{align*}
\text{read } (X,Y,Z); \\
\text{write } (Y,U,W) \\
\text{end.}
\end{align*}
\]

The reason is that no datum in the write-set of $T_5'$ depends
on the Y in the read-set. The problem can be resolved by forcing the reading of Y into an internal action, i.e., it is considered as part of the computation-phase of the transaction. This value is kept in a temporary area and is internally retrieved for further computation. After the transaction is completed, the value of Y is output to the database along with the other elements of the write-set. Thus, $T_5$ can be represented as

$$T_5'': \begin{align*}
\text{begin} \\
\text{read (X,Z);} \\
\text{write (Y,U,W)} \\
\text{end.}
\end{align*}$$

In general, the modeling of a multi-stepped transaction as a simple dual-stepped transaction may have a sizable impact on performance. For example, consider the transaction

$$T_6: \begin{align*}
\text{begin} \\
\text{read X;} \\
I = f(X); \\
\text{read Y[I];} \\
Z = g(Y[I]); \\
\text{write Z} \\
\text{end.}
\end{align*}$$
Since the value of $I$ is not known until run-time, the actual element of the array of $Y$ cannot be determined in order to form the static EDS, $(X, Y[I], Z)$. In this case, the simple dual-stepped transaction must take the form

$$\begin{align*}
T'_6 : \text{begin} \\
\quad \text{read } (X,Y); \\
\quad I = f(X); \\
\quad Z = g(Y[I]); \\
\quad \text{write } Z \\
\text{end.}
\end{align*}$$

The impact of this conversion on run-time efficiency depends on the size of $Y$ and the system load.

C. Basic Distributed Database Model

This section describes a basic distributed database (BDD) model for the concurrency control of transactions in a distributed database. Together with the foregoing framework, the model is to serve as a basis for the concurrency control algorithms studied in this research. The categorization of various concurrency control algorithms based upon such an underlying model will be uniform and easy to understand.
1. **Components of the BDD model**

There are two main aspects of the BDD model. These are the **logical components** and the **algorithmic framework** associated with the components. The logical components consist of

- transaction managers (TMs),
- data managers (DMs),
- data network (DN), and
- control network (CN).

The algorithmic framework is concerned with the transaction life-cycle.

2. **Logical components**

Only broad outlines are given for the functions of the various logical components. These outlines are subject to modification depending on the variations and details of individual concurrency control algorithms. The first two logical components, TMs and DMs, are based on an earlier model by Bernstein and Goodman (1981). Figure 3.1 graphically depicts the BDD model.

a. **Transaction manager**

A distributed database must have at least one transaction manager (TM). A TM is a logical entity and can be located physically anywhere in the system. Usually, a TM is an integral part of some DDBMS, although the number of TMs does not have to equal that of DDBMSs. A TM performs the following basic functions.
FIGURE 3.1. A Basic Distributed Database Model

1. Initial handling of each transaction according to some prearranged scheme (e.g. timestamping, static priority).
2. Decomposing a transaction into its component database actions, each of which can be handled by a DM.
3. Forwarding of actions to appropriate DMs and waiting for returned acknowledgements or data as
may be the case.

4. Supervision of the computation phase of the transaction.

5. Analyzing the effect of the transaction on the consistency of the database, either directly or indirectly, through the concurrency control algorithm.

6. Accepting or rejecting of a transaction depending on step 5.

7. Reinitiation of any rejected transaction.

b. Data manager

A data manager (DM) is a logical entity associated with each database site. Usually, a DM consists of two portions, each of which is an integral part of the DDBMS and DBMS of the site, respectively. The portion of a DM in a DDBMS handles distributed functions of a concurrency control algorithms, while that in a DBMS acts as an interface between distributed functions and the local database manager. As a whole, a DM performs the actions (accessing of data at its local database) for various transactions. The accessing of data is subjected to two concurrency control constraints. One of the constraints is the accessing protocol of the local database. The other is the global concurrency control protocol of the system. The nature of the local control is the same as in any centralized database, and thus is not considered part of the
concurrency control problem in a distributed database. The
global control will be dictated by the specific distributed
concurrency control algorithm.

c. **Data network** The data network (DN) is a logical
network which carries data (information to be used by the
transactions in their computation, and the result of the
computation to be stored in the database) between TMs and
DMs.

d. **Control network** The control network (CN) is
also a logical network but instead of carrying the data, the
network carries information involved with concurrency
control protocol of the system. Logically, the network
connects both the TMs and the DMs. The network may totally
or partially coincide with the data network.

It should be noted that the classification of a message
depends on its interpretation. For example, the
acknowledgements for write actions may be considered data
messages because they involve communication concerning the
data of the transactions. On the other hand, such a message
can be considered as a control message because it informs
the TM of the completion of another step in the process of
concurrency control algorithm. The exact classification of
the message types is of no concern in the categorization
scheme developed in this research.

Both the data network and the control network are
assumed to be reliable by many of the concurrency control algorithms. Both can have any topology, such as broadcasting or daisy-chaining organization, depending on individual algorithms. Also, many algorithms require that messages are pipelined, i.e. between any two sites, messages are delivered in the order sent. This research continues these assumptions since the focus of attention is on the categorization of given algorithms.

3. **Transaction life-cycle**

   Only simple dual-step transactions are allowed in the BDD model. Treatments of other types of transactions are not attempted here.

   A transaction goes through five logical phases during its life-time. The ordering and the overlapping of these phases depend on the specific concurrency control algorithm.

   1. **Initiation phase** A transaction, upon entering a TM, is stamped with necessary information according to the algorithm. For example, this information may be an internal system priority or possibly a timestamp. The data dictionary local to the TM is consulted to locate the DMs which contain the transaction's execution data set. The transaction is then broken in to its individual database actions.

   2. **Reading phase** The transaction's read actions
are forwarded to the target DMs via the data network. A DM, upon receiving a read action, performs necessary tasks according to the access protocol, and then returns the result to the originating TM. The result may be an acknowledgement, the required portion of the read set, or a reject message, depending on the algorithm.

3. **Computation phase**  The TM may start this phase when the necessary data has arrived, or it may wait until the read-set has been received.

4. **Write phase**  The result, obtained from the computation phase, is forwarded to target DMs in the form of write actions. Each DM then performs the necessary updates to its local database. Such updates do not alter the database state at this point. Instead, they are placed in nonvolatile but temporary storage. The DM then acknowledges the completion of the accessing. The forwarding of the write actions can be done in three ways.

- **Mono-batch write**  The TM waits until all values of the write set have been generated before sending the write actions.
- **Gradual write**  A write action is forwarded
as soon as a datum is generated by the computation phase.

- **Reservation write** The write actions are forwarded at the same time as the read actions. In this case, only the intent to write is forwarded without any actual data to be written (because they have not yet been generated). The DMs will reserve places in time for such write actions. The actual writing may be done by a mono-batch write or gradual write strategy. Both intent-to-write and write actions are acknowledged by the DMs.

5. **Termination phase** With information gathered through the control network, the TM may analyze and decide, with or without the help of DMs or other TMs, whether the transaction, if it were to be committed, would cause any inconsistency in the system. The transaction is then accepted or rejected accordingly. If the transaction is accepted, the TM informs all DMs involved to change the database state to permanently reflect the result of the transaction. This is similar to the concept of two-phase commit used by Gray (1978).
D. Summary

To summarize, a distributed database is simply a collection of database sites connected by a communication network. Each site contains a portion of the data in the system. Each datum is equally addressable through the use of the data dictionary, a copy of which resides at each site. A transaction submitted to a site, either locally or through remote submittal facility, must be simple dual-stepped and must in itself preserve the consistency of the database. These various components of the environment for concurrency control are summarized by the BDD model. It can be seen that the proposed model, although very simple and skeletal, is general and can accommodate commonly known variations of specific systems. Later on in the research, development of concepts will lead to more sophisticated models. However, the extended model will always retain the simplicity and general applicability of the proposed BDD model.
IV. DISCUSSION ON SERIALIZABLE EXECUTION SEQUENCES AND THEORETICAL BASIS FOR THE CATEGORIZATION SCHEME

Assuming the simple dual-stepped model of transactions discussed in Chapter III, this chapter establishes a theoretical basis for the categorization of concurrency control algorithms. This is done by investigating the properties of the set of execution sequences generated by the various concurrency control algorithms. Included are several abstract sets of execution sequences which have been studied by various other researchers. The result is the identification of a single large class of execution sequences, which includes the execution sequences generated by all of the other concurrency control algorithms. This containing class has important fundamental properties which serve as a theoretical basis in the ultimate categorization.

To guarantee the consistency of the database, it is necessary to impose certain restrictions on the concurrent execution of transactions. The concept of serializability is a widely accepted sufficient restriction. This concept also serves as the starting point in the search for a theoretical basis for categorization. Section A reviews this concept, repeating some definitions from Chapter I for completeness and extending the concept to reflect situations in a distributed database. Section B presents and simplifies the family of classes of serializable execution
sequences which preserve the consistency of the database. The analysis of this family reveals a certain class, the conflict-preserving-serializable class, the defining properties of which are common to all concurrency control schemes studied in this research. These defining properties are a basis for the categorization scheme presented in the next chapter. The analysis in Section B of this chapter is rather involved. Except for Subsection 5, which deals with the defining properties, it may be skipped on a first reading.

A. Serializability Concept

The concept of serializability was introduced by Eswaran et al. (1976). An execution sequence (a schedule, a log, or a history) of a group of transactions is the sequence of interleaved database actions from those transactions as processed concurrently by the system. This definition is used in practically all of the theoretical work in distributed databases and is universally accepted. However, this definition requires proper interpretation when used in a distributed system. The concept of merging local execution sequences into a single global execution sequence has been discussed by Bernstein et al. (1978) and it provides the basis for the needed interpretation.

First, consider the case of no redundancy. A local
execution sequence consists of the sequence of actions carried out by a DM at some site. A global execution sequence is the system-wide sequence constructed by merging local execution sequences at all sites subjected to the following constraints.

1. The ordering of actions in each transaction, which in this research is assumed to be a sequence, is preserved, and

2. the ordering of all actions in each local execution sequence is preserved in the global execution sequence.

Note that many merges (and consequently many global execution sequences) may be possible. This is perfectly acceptable since all such resulting global execution sequences will be equivalent, i.e. given an initial database state, the final states are all identical. Thus, by constructing a global execution sequence from local execution sequences (subject to the above two constraints), one actually selects a representative of an equivalent class.

Consider next the case of redundancy, that is, there are two or more copies of some data. In Chapter III, it was stated that two copies of a datum X will be considered distinct data X₁ and X₂ with the added consistency constraint that X₁=X₂. There are two cases to consider,
reading and writing multiple copies. It should be noted that the author is not aware of any method that actually reads multiple copies. The case is included here for completeness in the event that such an algorithm is found useful.

The following example illustrates the writing of multiple copies. It also illustrates the case where reading takes place on just one copy. Let $X_1$ and $X_2$ denote copies of the same datum. Suppose $X_1$ and $Y$ are located at site A and $X_2$ at site B. Assume the following two transactions

\begin{align*}
T_1 &: R_1[X_1]W_1[X_1,X_2] \\
T_2 &: R_2[Y]W_2[X_1,X_2]
\end{align*}

where R and W denote read and write actions, respectively. The subscripted numbers denote corresponding transactions. The symbols in brackets denote data to be acted upon. Two possible local execution sequences are

\begin{align*}
\text{site A} &: R_2[Y]R_1[X_1]W_1[X_1]W_2[X_1] \\
\text{site B} &: W_2[X_2]W_1[X_2]
\end{align*}

Two possible global execution sequences (obeying the two constraints) are

\begin{align*}
E_1 &: R_2[Y]R_1[X_1]W_1[X_1]W_2[X_1]W_2[X_2]W_1[X_2] \\
E_2 &: R_2[Y]W_2[X_2]R_1[X_1]W_1[X_1]W_2[X_1]W_1[X_2]
\end{align*}
Note that $E_1$ and $E_2$ are equivalent, that is the values of $X_1$ and $X_2$ produced in $E_1$ are the same as those produced in $E_2$. The value $X_1$ and $X_2$ may be different, in this case, because consistency has been violated.

The next example illustrates a reading of multiple copies. Using assumptions of the previous example, let $T_1$ be modified to reflect a reading of multiple copies.

$$T_1: R_1[X_1, X_2]W_1[X_1, X_2]$$

Two possible local execution sequences at the two sites are

- site A: $R_2[Y]R_1[X_1]W_1[X_1]W_2[X_1]$  
- site B: $W_2[X_2]R_1[X_2]W_1[X_2]$ 

Two possible global execution sequences are

$$E_1: R_2[Y]R_1[X_1]W_2[X_2]R_1[X_2]W_1[X_1]W_2[X_1]$$
$$E_2: R_2[Y]W_2[X_2]R_1[X_2]R_1[X_1]W_1[X_1]W_2[X_1]W_1[X_2]$$

Note that $E_1$ and $E_2$ are equivalent. There are, in this case, two read actions for $T_1$ and the first constraint for constructing a global execution sequence requires that both read actions must precede both write actions. An interesting consequence of this example is that the values of $X_1$ and $X_2$ read by $T_1$ may be unequal. This violates consistency of $X$, a subject which is discussed in greater detail in ensuing paragraphs.
An execution sequence is **serial** if its transactions are processed sequentially. The serializability concept can be defined as follows: An execution sequence preserves the consistency of the database (i.e., it is consistent) if it is equivalent to some serial execution sequence of the same set of transactions. Such an execution sequence is **serializable**. Serializability of the execution sequences is sufficient to preserve the consistency of the database. The following example illustrates the concept of serializability. Let \( T_1, T_2 \) and \( T_3 \) be three transactions which consist of the following database actions:

\[
\begin{align*}
T_1 & : R_1[X]W_1[Y] \\
T_2 & : R_2[Z]W_2[X] \\
T_3 & : R_3[V]W_3[Y,Z]
\end{align*}
\]

where \( R \) and \( W \) denote read and write actions, respectively. The symbols in brackets denote data to be acted upon. The execution sequence

\[
\]

is not serial while

\[
\]

is serial. However, the two are equivalent and thus \( ES_1 \) is serializable. The **serialization** of \( ES_1 \) is \( T_1T_2T_3 \). Note
that the serialization $T_1 T_2 T_3$ is not the only one for the three transactions that can preserve consistency through serializability. To illustrate this point, consider the execution sequence


which is equivalent to the serial execution sequence


Thus, $\text{ES}_3$ is serializable and preserves consistency. However, the serialization of $\text{ES}_3$ is $T_2 T_3 T_1$ which is different from that of $\text{ES}_1$.

As an example of nonserializability, consider the following execution sequence


Since $R_2$ precedes $W_3$ in $\text{ES}_5$, then $T_2$ would have precede $T_3$ in any equivalent serialization. Since $W_3$ precedes $W_1$ in $\text{ES}_5$, then $T_3$ would have precede $T_1$ in any equivalent serialization. But since $R_1$ precedes $W_2$ in $\text{ES}_5$, then $T_1$ would have similarly precede $T_2$. Because of this cyclic ordering, $\text{ES}_5$ cannot be equivalent to any serialization of the three transactions, and hence is not serializable. The term "precede" in this context refers to an ordering in real-time and is represented by the symbol "<". Precedence
in pseudo-time (e.g. that governed by timestamps), if used, will be explicitly mentioned in the accompanying context.

Unconstrained interleaving of actions may result in an inconsistent state of the database. A primary concern of concurrency control algorithms for distributed databases is to constrain the interleaving so that the resulting execution sequence is serializable.

The concept of serializability has been extensively analyzed by Bernstein et al. (1979) and Papadimitriou (1979). Their work is based on the concept of a conflict-graph. A conflict-graph (CG) is the representation of conflicts among possible transactions in the system over common data. The graph is a node-labeled, undirected graph, where the nodes correspond to database actions in the execution sequence. An edge in the graph connects either actions of the same transaction or the actions of different transactions which are in read-write or write-write conflict (i.e. there is a nonempty intersection between read-set and write-set or write-set and write-set, respectively). An example of a CG is shown in Figure 4.1. When a group of transactions interfere with each other such that there is an undirected cycle in their corresponding CG, it means that it is possible that some execution sequence(s) of this group of transactions is not serializable. If a CG contains no cycle, then the corresponding set of transactions cannot
cause nonserialzability when they are executed concurrently. An undirected cycle in a CG is usually denoted by an ordered n-tuple of the transactions involved. For example, Figure 4.1 contains a cycle \((T_1, T_2, T_3)\).

![Figure 4.1. An example of the conflict-graph](image)

**B. Aspects of Classes of Serializable Execution Sequences**

A number of classes of execution sequences have been proposed and studied by various authors. Some are derivable from known concurrency control algorithms. Others are interesting because of their abstract mathematical properties. In this section, these various classes are identified and their relationship is analyzed. This analysis results in the identification of a single class, the conflict-preserving-serializable (CPSR) class, which contains all but one of the other classes. This class provides defining properties which are used as a basis for
the categorization scheme developed in Chapter V.

The set of serializable execution sequences (SR) can be divided into a number of subclasses. Bernstein et al. (1979) have identified the following subclasses of SR:

- Conflict-preserving serializable (CPSR)
- Conflict-graph secure (CG-secure)
- Strictly serializable (SSR)
- SDD-1-secure
- Two-phase lock (2PL)
- Weak two-phase lock (W2PL)

Papadimitriou (1979) has identified three subclasses of SR:

- D-serializable (DSR)
- Q
- P3

It can be shown that

- CPSR, DSR and CG-secure are equivalent
- Q and W2PL are equivalent, and
- Q, 2PL, P3 and SDD-1-secure are proper subsets of CPSR.

Of these classes, the class SSR will be excluded from consideration because of the seemingly inherent inefficiency in determining if an execution sequence belongs to the class (Papadimitriou, 1979). There are some concurrency control algorithms which generate subclasses other than CPSR, e.g. 2PL (Menasce and Muntz, 1978 and Rosenkrantz et al., 1978)
and SDD-1-secure (Bernstein et al., 1978, Bernstein et al., 1980a and Rothnie et al., 1980).

1. Equivalence of DSR, CPSR, and CG-secure

An execution sequence is in the class conflict-preserving-serializable (CPSR) if and only if it is serializable and the order of conflicting actions is preserved in some equivalent serial execution sequence (Bernstein et al., 1979). Two execution sequences are clearly equivalent if they are identical except for the switching of a pair of consecutive and nonconflicting actions. An execution sequence is in the class D-serializable (DSR) if and only if it has some equivalent serial execution sequence which is obtained through a series of the above switchings (Papadimitriou, 1979). A projection of an execution sequence on a set of transactions is the execution sequence that is obtained by deleting, from the original execution sequence, the actions which belong to the transactions not in that set. For example, the projection of $R_1R_2W_2R_3W_1W_3$ over the set $(T_2,T_3)$ is $R_2W_2R_3W_3$. An execution sequence ES is in the class conflict-graph-secure (CG-secure) if and only if for every set of transactions that lies on a cycle in the conflict-graph (CG) of ES, the projection of ES on that set of transactions is in CPSR (Bernstein et al., 1979).
The equivalence of the classes CPSR and DSR has been pointed out explicitly by Bernstein et al. (1979). This equivalence can be conceptually illustrated by the definitions of the two classes. For an execution sequence to be in CPSR or DSR, the ordering of conflicting actions in the execution sequence must be preserved in some equivalent serial execution sequence. This preservation of ordering is explicitly defined in the case of CPSR, while in DSR the preservation is due to the switching of only nonconflicting actions to obtain the equivalent serial execution sequence.

The equivalence of classes CPSR and CG-secure is proved in the following theorem.

**Theorem 4.1.** The classes CPSR and CG-secure are equivalent.

**Proof.** The following two steps must be proved for an execution sequence ES.

- If ES is in CG-secure, then it is in CPSR.
- If ES is in CPSR, then it is in CG-secure.

The first part has been proved by Bernstein et al. (1979). For the second part of the proof, as suggested by Bernstein (1983), let $ES_i$ be an execution sequence which is not in CG-secure. By definition, $ES_i$ must have at least a projection which is not in CPSR. Since every projection of an execution sequence in CPSR must also be in CPSR, $ES_i$ is not in CPSR. Hence, if $ES_i$ is in CPSR, $ES_i$ is also in CG-secure.
2. **SDD-1-secure is a subset of P3**

Assume an arbitrary set of transaction $T_1, \ldots, T_n$, with a corresponding conflict-graph CG.

An execution sequence ES is in the class SDD-1-secure (Bernstein et al., 1979) if and only if, either

- there is no cycle in CG, or
- for all $T_i$ and $T_j$ in ES with the edge $(R_i, W_j)$ lying on a cycle in CG, $R_i < W_j$ in ES implies $W_i < W_j$ in ES.

For the class P3, the definition of a bad cycle is needed. A cycle $(T_1, T_j, \ldots, T_m)$ in CG is **bad** if and only if

- the write-set of $T_i$ intersects with either the read-set or write-set of $T_m$, and
- the read-set of $T_i$ intersects with the write-set of $T_j$.

In Figure 4.1, there are two bad cycles $(T_1, T_2, T_3)$ and $(T_2, T_3, T_1)$. Other cycles (such as $(T_2, T_1, T_3)$) are not bad.

It should be noted that for the class P3, the ordering of transactions in a cycle must be taken into account. This is different from the meaning of a cycle for the class SDD-1-secure.

An execution sequence ES is in P3 (Papadimitriou, 1979) if and only if, either

- there is no cycle in CG, or
- for all $T_i$ and $T_j$ in a bad cycle $(T_i, T_j, \ldots, T_m)$, $R_i < W_j$ in ES implies $W_i < W_j$ in ES.
Theorem 4.2. SDD-1-secure is a proper subset of P3.

Proof. Let ES be an execution sequence in SDD-1-secure. Let $(R_1, W_2)$ lie on cycle C in CG of ES and $R_1 < W_2$ in ES. Then by definition of SDD-1-secure, $W_1 < W_2$ in ES. If C is bad, then ES is in P3 by definition. If C is not bad, ES is in P3, since P3 has no constraint to exclude it. Hence, SDD-1-secure is a subset of P3.

To show proper containment, let ES be as follows.


The CG for ES is

Consider the cycle $C = (T_1, T_2, T_3)$ which is not bad. Note that $(R_1, W_2)$ is on C, $R_1 < W_2$ in ES, and $W_2 < W_1$ in ES. Thus, ES is in P3 but not in SDD-1-secure. Hence, SDD-1-secure is a proper subset of P3.

3. Equivalence of classes $Q$ and $W2PL$

An execution sequence ES is in the class $Q$ (Papadimitriou, 1979) if and only if each transaction $T_i$ in
ES can be assigned a serializability point $S_1$ such that

(a) $R_i < S_i < W_i$,

(b) if $R_i$ interferes with $W_j$ and $R_i < W_j$, then $S_i < S_j$, and

(c) if $W_i$ interferes with $W_j$ and $W_i < W_j$, then $S_i < S_j$.

The serializability point which is introduced here is an imaginary point and does not represent any database action.

An execution sequence ES is in the class W2PL (Bernstein et al., 1979) if and only if each transaction $T_i$ in ES can be assigned a lock point $P_i$ between $R_i$ and $W_i$ with the following properties.

(a) Suppose two transactions $T_i$ and $T_j$ are nested in ES, i.e. $R_i R_j W_j W_i$.

(1) If $P_i < P_j$, then $W_i$ does not interfere with either $R_j$ or $W_j$.

(2) If $P_j < P_i$, then $R_i$ does not interfere with $W_j$.

(b) Suppose two transactions $T_i$ and $T_j$ are intertwined in ES, i.e. $R_i R_j W_i W_j$.

(1) If $P_i < P_j$, then $W_i$ does not interfere with $R_j$.

(2) If $P_j < P_i$, then $W_j$ does not interfere with either $R_i$ or $W_i$.

The nature of a lock point is the same as that of a serializability point.

**Theorem 4.3.** The classes Q and W2PL are equivalent.
Proof. To prove that the classes Q and W2PL are equivalent, the sets of allowable execution sequences of both classes are compared. If they are equal, then Q and W2PL are equivalent.

Let $E_{Sp}$ be the projection of some execution sequence $ES$ over $(T_i, T_j)$ in $ES$. Any such $E_{Sp}$ can be represented by a 6-bit binary number. Each bit denotes a property that $E_{Sp}$ can have. The description of each bit is as follows.

- **Bit (i):** ordering of $S_i$ and $S_j$ (or $P_i$ and $P_j$).
  
  0 denotes $S_i < S_j$.
  
  1 denotes $S_j < S_i$.

- **Bit (ii):** ordering of $R_i$ and $W_j$.
  
  0 denotes $R_i < W_j$.
  
  1 denotes $W_j < R_i$.

- **Bit (iii):** ordering of $W_i$ and $W_j$.
  
  0 denotes $W_i < W_j$.
  
  1 denotes $W_j < W_i$.

- **Bit (iv):** interference between $R_i$ and $W_j$.
  
  0 means no.
  
  1 means yes.

- **Bit (v):** interference between $W_i$ and $R_j$.
  
  0 means no.
  
  1 means yes.

- **Bit (vi):** interference between $W_i$ and $W_j$.
  
  0 means no.
1 means yes.

Figure 4.2 enumerates $E_{p}$ that are allowed (denoted by "A"), not allowed (denoted by "N") and conditionally allowed (denoted by "C"). The terms "allowed" and "not allowed" mean $E_{p}$ is always in or always not in the specified class, respectively. The term "conditionally allowed" refers to $E_{p}$ that has the following properties: $S_{i} < S_{j}$, $R_{i} < W_{j}$, and $W_{i} < W_{j}$. Two execution sequences are possible: $R_{i} W_{i} R_{j} W_{j}$ and $R_{i} R_{j} W_{i} W_{j}$. The first sequence is serial, and hence no restriction is imposed. The second sequence is possible if $W_{i}$ does not interfere with $R_{j}$.

Since the two tables of enumeration are identical, the classes $Q$ and W2PL are equivalent.

4. Implications of the classes CPSR and SSR

The previous discussion refines relationship among subclasses of SR which has earlier been forwarded by Bernstein et al. (1979) and Papadimitriou (1979). The refinement is shown in Figure 4.3. Note that SSR is the only subclass which is not contained in CPSR. Papadimitriou (1979) has observed that it is not known whether the class SSR is efficiently recognizable. If this is true, then practical concurrency control algorithms could not generate execution sequences that are in the class SSR but not in CPSR. This section supports Papadimitriou's contention by exhibiting those properties of an execution sequence that
FIGURE 4.2. Allowable execution sequences of classes Q and W2PL

cause the difficulties. As a result, the class SSR will be excluded from further study in this research.

a. Notions of liveness, deadness and detailed transactions

Prior to giving a precise definition of SSR, it will be helpful to formulate the notions of live and dead actions and transactions, and the concept of a detailed transaction.

Let an execution sequence ES be augmented by imaginary
FIGURE 4.3. Relationship among subclasses of serializable execution sequences

input and output transactions, $T_{in}$ and $T_{out}$ (Bernstein et al., 1979), where

- $T_{in}$ consists of an empty read-set and a write-set which is the union of all the read-sets in ES,
- $T_{out}$ has an empty write-set and a read-set which is the union of all the write-sets in ES,
- $T_{in}$ precedes all other transactions, and
• $T_{out}$ follows all other transactions.

A read action $R_i$ is said to read the value of a datum $X$ from a write action $W_j$ of another transaction if $W_j < R_i$ in augmented ES and for all $W_k$ such that $W_j < W_k < R_i$ in augmented ES, $W_k$ does not involve $X$. A live transaction is defined as follows.

• $T_{out}$ is live.

• If one or more of the data items in the write-set of $T_i$ in ES is read by a live transaction in augmented ES, then $T_i$ is live in ES.

A dead transaction is the transaction which is not live. It should be noted that liveness is defined to be limited to the selected execution sequence. Liveness may or may not be globally true if subsequent execution sequences are taken into account.

Usually, a write action involves more than one datum. A dead action is a write action in which none of its data are read. A live action is a write action that is not dead. It may be that some data of a live action is not read by another action. Thus, in order to simplify the problem, an action with more than one datum is broken down into subactions so that the effect of each datum can be examined separately. The following definitions concern the breaking down of such a type of action. A detailed transaction is a transaction in which each of its actions involves only one
datum. A transaction is said to be represented in detailed form if each of its actions $A_i[X_1, X_2, \ldots, X_n]$ is replaced by $A_i[X_1] A_i[X_2] \ldots A_i[X_n]$ where $A_i$ is either $R_i$ or $W_i$. Note that if an execution sequence is modified by replacing each of its transactions by the detailed form of that transaction, then the new execution sequence thus modified is still equivalent to the original one. A detailed execution sequence consists only of transactions that are either detailed or are represented in detailed form. In a detailed execution sequence, a dead action is a write action, the datum of which either is not read at all or is read only by dead transactions. A live transaction must have at least one live action, while a dead transaction has only dead actions. A consequence of the definition of a dead action is that a dead action in an execution sequence must conflict with at least one other subsequent write action in that execution sequence. This is because if a write action does not conflict with any other subsequent write action, then its data must be eventually read by $T_{out}$ and thus would be live. The deadness of an action can be considered as the overwriting of the datum involved by some following write action which conflicts with the dead action. Such an overwriting occurs before the datum can be read by some read action.

In subsequent sections concerning the dead actions in
this chapter, it is assumed that all actions are detailed or are represented in detailed form.

b. Subclasses of SR concerning live actions  In this section, two subclasses of SR are introduced. The major characteristic common to the two classes is that they involve only live actions. Examination of the two classes suggests the reason why the class SSR may not be efficiently recognizable.

An execution sequence is said to be in the strong-SR class if and only if it is serializable and contains only live actions.

**Theorem 4.4.** The class strong-SR is properly contained in the class CPSR.

**Proof.** If a detailed execution sequence is in strong-SR, then any $W_i$ must have at least one interfering $R_j$ which reads from $W_i$ (the read action may belong to $T_{out}$). This read-from relationship is reflected in the partial ordering of transactions. Thus, the ordering of $W_i$ and $R_j$ is the same as in any equivalent serial execution sequence and as a result the order of every pair of conflicting actions is preserved. Hence, the execution sequence is in CPSR. The strong-SR is properly contained in CPSR by the fact that there exist execution sequences in CPSR that contain dead actions.
An execution sequence ES is in the class strictly-serializable (SSR) if and only if it is serializable and for any pair of transactions $T_i$ and $T_j$, if $W_i < R_j$ in the ES, then $W_i < R_j$ in any equivalent serial execution sequence (Bernstein et al., 1979). An execution sequence is in the class strong-SSR if and only if it is in SSR and its detailed form contains no dead actions.

Theorem 4.5. The class strong-SSR is properly contained in the class strong-SR.

Proof. By definition, any execution sequence in strong-SSR is also in strong-SR. The reverse is not true because there are execution sequences that are in strong-SR but not in strong-SSR. For example, the execution sequence


has an equivalent serial execution sequence $T_3T_1T_2$. Since all actions are live, it is in strong-SR. However, the ordering of $T_2$ and $T_3$ is switched so it is not in SSR and thus not in strong-SSR. Also, it follows directly from the definitions that the class strong-SSR is the intersection of the classes strong-SR and SSR.

Refering to Figure 4.3, the relationship among the classes SR, CPSR, SSR, strong-SR and strong-SSR can be represented by the Venn diagram in Figure 4.4. The following execution sequences exemplify the class
containment of Figure 4.4.

- In SR - CPSR - SSR:
  \[ W_2[Z]W_4[Y] \]

- In SSR - CPSR:

- In CPSR - (strong-SR) - SSR:

- In CPSR - (strong-SR) U SSR:

- In (strong-SR) - SSR:

- In strong-SSR:

c. On the possibly inefficient recognition of members of SSR

One reason why an execution sequence in SSR may not be in CPSR can be illustrated through the manipulation of dead actions. Recall that a dead action is a write action that must conflict with some subsequent write action. Since a dead action is not involved in the consideration of data dependencies, its relative position in any execution sequence is not reflected in the partial ordering of the transactions of that execution sequence. Suppose, in the class SSR, an execution sequence ES with a dead action is equivalent to some serial execution sequence but with the
The position of the dead action changed in the serial execution sequence such that

- the dead action remains dead,
- the dead action does not change any data dependency of the original ES, and
- the ordering in ES between the dead action and some conflicting action is reversed.

Then, the CPSR constraint is violated but the partial ordering of transactions remains the same. Hence, the execution sequence is in SSR but not in CPSR. As an example, consider the following set of transactions.
The execution sequence

\[ T_1 : R_1[V]W_1[X] \]
\[ T_2 : R_2[X]W_2[X] \]
\[ T_3 : R_3[Z]W_3[X] \]

is constructed over the three transactions. The equivalent serialization is \( T_1T_3T_2 \). The execution sequence is in SSR because the sequence \( T_3T_2 \) is preserved in the serial sequence. It is not in CPSR because the ordering of \( W_1 \) and \( W_3 \) is reversed.

From Theorem 4.4, 4.5 and the above argument, it can be seen that if dead actions are excluded from the execution sequences, the classes SR and SSR are reduced to the classes strong-SR and strong-SSR, respectively. The resulting two classes are properly contained in the class CPSR. These containments are due to the possibility of dead actions in the class CPSR. Nevertheless, the definition of CPSR does not allow manipulation of these dead actions, so as not to cause the reversal of the ordering of conflicting actions. Thus, the analysis suggests that the exclusion, or at least the restricted movement, of dead actions may account for the efficient recognizability of the class CPSR. Since the manipulation of dead actions is an inherent property of the class SSR, the class may not be efficiently recognizable.
5. **CPSR class as a basis for categorization**

The preceding analysis indicates that the class CPSR is an appropriate theoretical choice on which to base a scheme of categorization. There are two reasons for this. First, any class, not contained in the CPSR class, might have undesirable practical implications. For example, SSR allows the manipulation of dead actions. As a result, it is an open question whether or not an efficient technique can be devised to recognize members of that class. This would preclude the existence of a practical concurrency control algorithm. Second, no proper subclass of the class CPSR, which has been studied by this author and others, contains all of the other interesting proper subclasses.

**C. Summary**

The discussion in Section B produced a simplified version of the family of classes of serializable execution sequences. This simplification indicates an important class, the conflict-preserving-serializable (CPSR) class, the defining properties of which are employed as a basis for the categorization scheme developed in Chapter V.
V. THE CATEGORIZATION SCHEME

A categorization scheme for concurrency control algorithms is presented in this chapter. Section A states a pragmatic reason for selecting the class CPSR, the defining properties of which serve as a basis for the categorization. This reason for the selection complements the other two theoretical reasons presented in the preceding chapter. Section B presents two philosophical approaches to the concurrency control problem. These philosophies define the first level of categorization. Section C and D present the remaining levels of the categorization tree. Individual concurrency control algorithms are classified in their appropriate categories in Section E.

A. A Pragmatic Reason for Selecting CPSR Class

The previous chapter gave two theoretical reasons for selecting the class CPSR as the basis for categorization of concurrency control algorithm. A third reason, which is pragmatic in nature, is that each concurrency control algorithm, studied by this author, appears to satisfy the defining properties of CPSR. This would imply that the set of execution sequences generated by each algorithm is a subset of CPSR. A formal proof of this observation for every concurrency control is difficult to obtain due to at least two reasons.
1. Not all algorithms have been implemented and, as a consequence, their correctness has not been established (Bernstein and Goodman, 1981).

2. The details of implementation add much complexity to the written description, obscuring in some cases the components directly involved in concurrency control and, in one case (Ellis, 1977), obscuring the precise analysis performed for guaranteeing consistency.

For each concurrency control algorithm, the author assumes correctness of the method. The precise analysis for insuring consistency, in cases where the description is vague or is left open, is assumed to be based either on pure syntax or on a semantic analysis which would not produce execution sequences outside of the class CPSR. The latter assumption is necessary to insure that the algorithm is not specialized for a specific database, and as a consequence, of little interest in this research. The categorization scheme, proposed in this chapter, attempts to abstract out the details of implementation by starting with the defining properties of the CPSR class as the theoretical basis for categorization.

The following example illustrates how one might approach the problem of formally proving that a concurrency control method generates execution sequences only in the
CPSR class. This illustration is based on the concurrency control algorithm proposed by Ellis (1979). The algorithm can be viewed as a locking technique. The initiating site of a transaction acquires a lock on its local copy of the read-set prior to the computation. After the computation, it acquires a lock on each copy of its write-set before the update can be applied to the database. Unanimous consensus (i.e. all locks requests must be granted) is required for the initiating site to proceed with the update, else the transaction is rejected to avoid possible inconsistency.

The locking procedure on a transaction's write-set is theoretically done instantaneously, and a lock must either be acquired at each site or none can be acquired. Actually, delays in the communication network render impossible this instantaneous acquiring of all locks. Thus, the locks on write-set are obtained in some sequential order. Also, all locks are released after the completion of the transaction. As a result, the actual execution sequence produced by this algorithm is a special case of the class 2PL since all locks are released after the completion of the transaction (Eswaran et al., 1976). (An execution sequence is in 2PL if and only if all of its transactions are two-phased. A two-phased transaction is defined as a transaction that does not lock any more datum if it has unlocked some datum.) From the discussion in the previous chapter, it is known that 2PL
is a subset of CPSR. Thus, the illustration could be concluded at this point. However, in the spirit of illustrating that Ellis-type concurrency control algorithms satisfy the two defining properties of the class CPSR, the discussion is further extended. Recall the two defining properties of the class CPSR.

1. There exists a partial ordering $\mathcal{O}$ among transactions.

2. The ordering of any pair of conflicting actions is the same as the ordering in $\mathcal{O}$ of their respective transactions.

Eswaran et al. (1976) has shown that 2PL execution sequences are serializable. So the first defining property is satisfied. For the second defining property, consider a database site $A$. Let $T_i$ and $T_j$ be transactions local and remote to $A$, respectively. It suffices to show that if $T_i$ conflicts with $T_j$ then the ordering of their actions at all sites must be the same as the ordering of $T_i$ and $T_j$ in some partial ordering of transactions. Since $T_j$ is remote to $A$ and the database is fully redundant, $A$ sees only $W_j$ and not $R_j$. There are two cases where $T_i$ can conflict with $T_j$ at site $A$.

1. The intersection of the read-set of $T_i$ and the write-set of $T_j$ is not empty.

2. The intersection of the write-sets of the two
transactions is not empty.

For the first case, suppose \( \text{read-set}_i \cap \text{write-set}_j \neq \emptyset \).

Either \( W_j < R_i \) or \( W_i < W_j \) because \( \text{read-set}_j \) is locked for the duration of \( T_i \). If \( W_j < R_i \) at \( A \) then \( W_j < W_i \) at \( A \). From the properties of the two-phase locking technique, \( W_j < W_i \) at all sites. In either case, the effect of \( T_i \) on the database relative to \( T_j \) must be the same, at all sites, as in some equivalent serial execution sequence. A similar argument can be used for the second case where \( \text{write-set}_i \cap \text{write-set}_j \neq \emptyset \).

Rather than attempting a proof that each method is in the class CPSR, the author identifies those features of an algorithm that enforce the defining properties of the class CPSR. In the categorization scheme, the defining properties of the class CPSR may appear at various levels of the tree, and one or more categories may be involved in exhibiting each defining property.

B. Philosophy of Primary Categories

Although the discussion in the foregoing section indicates that every concurrency control algorithm studied in this research employs the two defining properties of the class CPSR as its basic algorithmic component for producing execution sequences, the algorithms express these defining properties in different ways. Nevertheless, there are
certain logical, and systematic, approaches in expressing these defining properties. These approaches, presented in this section, are the basis for two primary categories of concurrency control, namely prevention and correction.

1. Logical approaches to concurrency control problem

There are two logical approaches to ensuring serializability of execution sequences. The first approach is prevention where the processing of database actions is constrained to always yield serializable execution sequences. The other approach, correction, examines a resulting execution sequence and corrects or adjusts the sequence when nonserializability occurs. These two basic approaches to the problem, together with the defining properties of the class CPSR, form a set of philosophies in solving the concurrency control problem.

a. Prevention of nonserializability

The philosophy of prevention can be broken down into two steps.

1. Construction of an ordering (either partial or total) of transactions, termed a prescription.
2. Enforcement of the prescription on actions so that conflicting actions are ordered in the same way as their corresponding transactions.

Employing this philosophy, the system will process the actions in the prescribed sequence. This will unerringly yield serializable execution sequences.
b. **Correction of nonserializability**

The philosophy of correction can be expressed as follows.

1. A transaction is tentatively executed by the system without any restriction. The database state is not changed at this point.
2. Conflicts among transactions are distributedly observed.
3. An analysis is made based on the ordering of conflicting actions to detect any nonserializability.
4. If nonserializability is found, it is remedied (usually through the roll-back of some transaction). Otherwise, the database state is updated.

2. **The role of transaction rejection**

The concept of prevention as presented in the previous section would seem to imply that all aspects of any undesired activity is identified before the fact and not allowed to occur. Such an interpretation is too constraining for an appropriate categorization of concurrency control algorithms. This research interprets prevention to mean that no part of the processing phase of the transaction is allowed to occur. In terms of points of roll-back, in the life-cycle of the transaction model defined in Chapter III, the following definition are
adopted.

1. A concurrency control algorithm, based on prevention, may or may not involve any roll-back. If roll-back is needed it must always occur before the computation-phase.

2. A concurrency control algorithm is based on correction if, for at least one transaction, roll-back can occur during or after the computation-phase.

Roll-back after the read-phase of a transaction wastes only message exchanges involved in the acquisition of the read-set (some processing is also required for housekeeping and data dictionary look-up, but this is considered less significant). On the other hand, a roll-back during or after the write-phase means wasting the resources used by the read-phase, the computation-phase, and the write-phase.

The definitions of the terms "prevention" and "correction" as suggested here, may differ from those proposed by other researchers.

3. Distinction between subcategories and components

In the categorization scheme described in this chapter, a category (or subcategory) has one or more components and each component has one or more subcategories. The components of a category are the various disjoint parts that comprise the category. The subcategories of a component are
variations in techniques in which the component may be realized.

C. Subcategorization of Prevention

The philosophy of prevention indicates that there are two major components in every concurrency control algorithm which belongs to this category. The first component constructs the prescription of the ordering of action-processing in such a way that if the system processes actions according to the prescription, the resulting execution sequence will always be serializable. This component is termed the preanalyzer. The second component for prevention is the enforcing of the prescription, and is termed the enforcer. These two components directly reflect the two defining properties of the class CPSR. Table 5.1, presented at the end of this section, summarizes the relationship among components and subcategories of prevention to be proposed and discussed.

1. Preanalyzer

A preanalyzer must specify a partial ordering of transactions. The technique in establishing the partial ordering can range from the simplest, such as the multi-queued first-come-first-served priority scheme employed in job scheduling of conventional operating systems (Deitel, 1983), to a complex use of semantics of transactions.
The use of semantics requires a great deal of processing power in the preanalyzer and may not be efficient. An apparent disadvantage is that the use of semantics may not be easily automated. This puts additional burden on the users. Although the use of semantic analysis can result in execution sequences outside the class CPSR or even ones that are not serializable, the effective use of such an approach has not been demonstrated. As a result, a preanalyzer is assumed to employ only information that can be effectively automated. Furthermore, it is assumed that such a preanalyzer will not produce execution sequences outside the class CPSR.

A preanalyzer based on the syntax of transactions is called a **simplified preanalyzer**. A simplified preanalyzer consists of two components: organization and provision for concurrency. Simplified preanalyzers may be organized centrally or distributedly. The provision for concurrency allows for a partial, rather than total, ordering of transactions. This feature must be distinguished from the concurrency provided in the enforcer, the nature of which is described later.

a. **Organization of simplified preanalyzer**

A simplified preanalyzer may either be centrally or distributedly organized.
1) **Centralized simplified preanalyzer**

Usually, this form of preanalyzer represents the simplest technique in concurrency control, where only one transaction manager is involved. This central site accepts transactions from all the other sites, processes them according to some priority (usually on a first-come-first-served basis), and broadcasts the write actions or updates to all sites. Consecutive sequence numbers are tacked onto updates so that sites can apply the updates in the order specified by the central site. In such a system, the database may range from no redundancy to full redundancy. The redundancy serves as back-up in case of failures (where the central control can move to another site). Examples of such schemes are the complete centralization algorithm (CCA) and centralized locking algorithm (CLA) proposed by Garcia-Molina (1979). This same work also suggests a number of variations on the basic centralized scheme. A more complex form of centralized control is one where the control can move from site to site even in the absence of failure. This concept is termed **migrating central control**. The migration can either be effected by a unanimous consensus of all participating sites (Seguin et al., 1979), or by forcing the central control along a virtual ring based on some time-out mechanism (Greene, 1981).
2) **Distributed simplified preanalyzer**

Usually, in a distributed simplified preanalyzer, a system of distributed sequencers is used to generate a totally ordered set of timestamps. These timestamps are attached to the transactions, thus creating the desired prescription. The prescription in this case is simply a total ordering of all known transactions in the system. There are various types of timestamping schemes. One is the concept of consecutive timestamps in which a set of consecutive nonnegative integers is generated. One way to distribute this function is through the use of a circulating token (Le Lann, 1978) where each site takes its turn in generating some consecutive portion of integers. Another method, which is widely used, generates nonconsecutive timestamps (Lamport, 1978). In this later method, a timestamp consists of two parts. The first part gets its value from a consecutive counter associated with the originating site of the timestamp. The second part is a unique number associated with each site. This second part is used as a tie-breaker for those timestamps of different sites that have identical first parts. The fairness of this scheme has been questioned. A possible remedy is the MOD numbering scheme proposed by Rahimi and Franta (1982).

b. **Provision for concurrency**

A simplified preanalyzer usually constructs a total ordering of
transactions. This in itself does not degrade the degree of concurrency of the whole system, since the system is multiprocessin in nature through its distribution, and the concurrency in using the system database is limited by the concurrency in using the local database at each site. The degree of local concurrency is in turn determined by the architecture of the local database manager which is transparent to the concurrency control protocol of the system. The drawback of a total ordering of transactions is that the concurrency control protocol at each site must select the next action to be processed by the local database manager in a strictly sequential manner. A sequential form of selection can delay actions of those transactions which are younger but do not conflict with the actions of older transactions. Relaxing this bottleneck into a partial ordering of transactions can be realized in two ways;

- Dynamic conflict-analysis
- Static conflict-analysis

1) Dynamic conflict-analysis In this approach, the partial ordering is created after the total ordering has been established. This partial ordering is based on conflicts among transactions and results in some form of directed acyclic graph. This concept is centrally oriented and can take many forms as suggested by Garcia-Molina (1979). Distribution of dynamic conflict-analysis is
generally considered infeasible (Papadimitriou, 1979).

2) Static conflict-analysis In this approach, restrictions are imposed on the manner in which conflicts can occur. Unlike the dynamic case, algorithms which depend on static conflict-analysis can be distributed. The restriction on conflicts may be achieved through transaction aggregation where transactions are grouped into classes at the time of system design. In this case, a set of protocols based on conflicts among these classes serves to partially order transactions at run time. Such an approach is employed in the SDD-1 system (Bernstein et al., 1980a and Rothnie et al., 1980). Another form of static conflict-analysis is data partitioning. In this case, the total ordering of transactions is relaxed to a partial ordering by allowing actions on separate partitions to be processed concurrently. An example of such an approach is the partitioned distributed data sharing system proposed by Le Lann (1978).

2. Enforcer

After the preanalyzer of the system has prescribed an ordering of transactions, the enforcer accepts the prescription and makes sure that the database actions are processed according to the prescribed ordering. This is carried out by a component of the enforcer termed the oldest-action-selection (OAS). After the preanalyzer has
constructed a serializable prescription, each action will have a set of predecessors and a set of successors. Each action must be processed after all of its predecessors and before all of its successors. Thus, for a group of actions appearing at a data manager (DM), a certain action can be selected as the next action to be processed if and only if all of the action's predecessors have been processed by that DM, and the action is the predecessor of all other actions currently presented or to be presented at the DM. In other words, the action is the "oldest" of all actions that have not been processed at that DM (whether they have been received or not). It should be noted that if the system allows for a partial ordering of transactions, the actions from unordered transactions may be processed in parallel without regard to which is the oldest. The complexity of an OAS depends on whether consecutive or nonconsecutive timestamps are generated by the preanalyzer.

a. OAS with consecutive timestamps  
In the case of consecutive timestamps, it is a simple matter for the OAS to select the next appropriate action by just waiting for the action with the next higher timestamp to arrive. In this case, timestamps are usually called sequence numbers. A system with this type of OAS does not allow for any rejection of read actions because of the strict sequencing of timestamps.
Since timestamps are consecutive, concurrency among actions must be determined from the partial ordering of transactions explicitly specified by the preanalyzer, or it must be determined by the database manager local to each DM. In the first case, the preanalyzer provides a separate sequence of timestamps for each independent portion of data. An example is the partitioned DDSS which uses tickets (Le Lann, 1978). More complicated schemes which provide concurrency are those which involve a centralized locking scheme (Garcia-Molina, 1979). For the second case, the DM forwards, one by one, actions which have been granted processing by the OAS to the local database manager. The local database manager may exploit parallelism provided by the architecture to speed up processing.

b. **OAS with nonconsecutive timestamps** If the timestamps are nonconsecutive, the OAS, without some form of constraint, cannot be sure that there are no missing actions in the ordering of timestamps presented at the DM at a given instant. There are two constraints proposed by Lamport (1978) to ensure proper selection of each oldest action.

- Actions sent and received between a TM and a DM must be pipelined, i.e. they are in timestamp order.
- An action is deemed oldest at a DM, if it is the earliest of all pending actions at the DM and the
DM has received a later timestamp from each TM in the system.

OAS with nonconsecutive timestamps contains four components:
- OAS for read actions
- OAS for write actions
- Reservation of write actions
- Provision for local concurrency

1) OAS for read actions Usually, the read and write actions received at a DM are ordered by the same stream of timestamps. But in some systems updating of a database takes precedence over reading. The SDD-1 algorithm by Bernstein et al. (1978) contains a number of protocols for different classes of transactions. One such protocol, the protocol P1, concerns read-only transactions. A TM can arbitrarily selects a timestamp for the read actions of a read-only transaction. The selection enables the actions to be processed by DMs more quickly. The drawback is that such a read action can arrive at a DM after some younger write action (in terms of timestamp) has been processed. In this case, the read action has to be rejected. (Note that pipelining in such a system may not hold).

2) OAS for write actions The OAS can select write actions in a straightforward way. But, as pointed out by Thomas (1978), write actions do not have to be processed in timestamp order as long as an older write action does not
supersede a younger one which has already been processed. This is called the Thomas Write Rule (TWR). In this case, an out-of-order write action is not rejected. Instead, it is simply ignored. This technique usually requires a timestamp on each addressable datum. The timestamp reflects the timestamp of the latest write action which updates the datum. A write action is out-of-order if its timestamp is older than that of the datum to be updated.

3) Reservation of write Usually, the write actions of a transaction are forwarded to DMs when the computation phase of the transaction concludes. This can result in a bottleneck at the TM because transactions have to be processed sequentially due to the pipelining constraint. Another strategy is to reserve write actions. Reservations of write actions are forwarded (without their values) to DMs at the same time as read actions. Hence, the TM is immediately free, after the read-phase, to handle the next transaction. The DM is also free to handle the messages of the next transaction, thereby allowing for additional concurrency. In a system with reservation of write actions, as in the algorithm by Milenkovic (1980), the OAS at a DM will act on a write action at the time when its computed value is actually received. A write action is considered out-of-order if it receives its computed value later than some younger write action. The reservation of
write actions serves as a template into which read actions can fit, that is, a read action will not be processed until the write action with the latest earlier timestamp has been executed. As a result, there is no need to reject any read action.

For a system with multi-traced transactions (see Chapter III for multi-traced transactions and the associated discussion), reservation of write actions implies the use of static EDS. This, as has been noted, may degrade performance depending on the size of the EDS and system load.

4) Provision for local concurrency If the degree of concurrency at the local database site depends only on the partial ordering of transactions provided by the preanalyzer, the OAS is considered passive, i.e., it follows the type of ordering dictated by the preanalyzer. On the other hand, an OAS may relax the ordering provided by the preanalyzer, based on the granularity of the local database. This is the subqueue mechanism proposed earlier by Milenkovic (1980), and also refined in the preventive CPSR algorithm proposed in Chapter VI. The OAS with such a provision is considered active because it modifies the ordering prescribed by the preanalyzer. Actions in separate subqueues are allowed to proceed concurrently.
3. **Categorization tree for prevention**

The categorization of prevention is tabulated in Table 5.1. The labeling scheme for the various subcategories will be explained in a later section.

D. **Subcategorization of Correction**

Three components can be identified for every correction algorithm. The first two components are related. The first component deals with conflict-analysis and the second deals with waiting requirement which, if present, manifests itself in terms of locking. Thus, if a waiting requirement is present, the conflict-analysis is actually deadlock detection and correction. If a waiting requirement is not present, the conflict-analysis deals with nonserializability and its correction. The term nonserializability will be used in a generic sense to refer to either the usual nonserializability or the presence of deadlock. Similarly, serializability will assume its normal meaning in the nonlocking algorithms or the absence of deadlock in the locking algorithms. The third component, independent of the other two, deals with the provision for local concurrency. Table 5.2, presented at the end of this section, summarized the relationship among components and subcategories of correction which are to be presented and discussed in this section.
<table>
<thead>
<tr>
<th>Subcategories for Prevention</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A: Prevention</strong></td>
</tr>
<tr>
<td>* Preanalyzer</td>
</tr>
<tr>
<td>• A: Complex preanalyzer</td>
</tr>
<tr>
<td>• B: Simplified preanalyzer</td>
</tr>
<tr>
<td>* Organization</td>
</tr>
<tr>
<td>• A: Centralized</td>
</tr>
<tr>
<td>• A: Fixed central site</td>
</tr>
<tr>
<td>• B: Migrating central control</td>
</tr>
<tr>
<td>• B: Distributed</td>
</tr>
<tr>
<td>• A: Consecutive timestamp</td>
</tr>
<tr>
<td>• B: Nonconsecutive timestamp</td>
</tr>
<tr>
<td>* Provision for concurrency</td>
</tr>
<tr>
<td>• A: Total ordering</td>
</tr>
<tr>
<td>• B: Partial ordering</td>
</tr>
<tr>
<td>• A: Dynamic conflict-analysis</td>
</tr>
<tr>
<td>• B: Static conflict-analysis</td>
</tr>
<tr>
<td>• A: Transaction aggregation</td>
</tr>
<tr>
<td>• B: Data partitioning</td>
</tr>
<tr>
<td>* Enforcer</td>
</tr>
<tr>
<td>• A: OAS with consecutive timestamp</td>
</tr>
<tr>
<td>• B: OAS with nonconsecutive timestamp</td>
</tr>
<tr>
<td>• OAS for read action</td>
</tr>
<tr>
<td>• A: Nonrejectable read</td>
</tr>
<tr>
<td>• B: Rejectable read</td>
</tr>
<tr>
<td>• OAS for write action</td>
</tr>
<tr>
<td>• A: Strict ordering</td>
</tr>
<tr>
<td>• B: Thomas Write Rule</td>
</tr>
<tr>
<td>• Reservation of write action</td>
</tr>
<tr>
<td>• A: None</td>
</tr>
<tr>
<td>• B: Allowed</td>
</tr>
<tr>
<td>* Provision for local concurrency</td>
</tr>
<tr>
<td>• A: Passive OAS</td>
</tr>
<tr>
<td>• B: Active OAS</td>
</tr>
</tbody>
</table>

The labeling scheme for components and subcategories is explained in Section E.
It should be noted that the two defining properties of the class CPSR do not correspond to first level components in correction. Instead, they are subcomponents of the main component conflict-analysis.

1. Integrity of conflict-analysis

The analysis of conflicts can either be integral or fractional, depending on the extensiveness of the conflict-information available.

a. Integral conflict-analysis In this approach, conflicts among transactions detected at various DMs are analyzed for serializability, usually by some centrally-located analyzer in a TM. At a given instant, the system may contain a number of nonoverlapping conflict-graphs of transactions. The integral analysis requires a complete knowledge of conflicts among all transactions in some disjoint portion of the conflict-graph. Such an integral knowledge does not necessarily imply total knowledge of all conflicts. There are three components associated with the integral analysis. The first is the multiplicity of the analyzer. The second component is the way in which the information concerning the conflicts is forwarded to the analyzer. The last component is the precedence selection which aids in the selection of the transaction to be rolled back in the case of nonserializability.
1) Multiplicity of integral analyzer

The information on conflicts among transactions at DMs can be forwarded to a single common site where the analysis of conflict can be carried out (Menasce and Muntz, 1978, Stonebraker, 1978 and Dewitt and Wilkinson, 1980). This concept of mono-analyzer handles conflict-information in term of system as a whole (i.e. it is system oriented), and requires all information on conflict. However, there are two drawbacks to such an arrangement. The first is that the load on the central analyzer is high, creating a bottleneck. The other drawback is the lack of redundancy, which may result in less reliability. The remedy is to have a multi-analyzer, one for every currently executing transaction. In this scheme, each DM forwards conflict information between transactions back to their originating TMs, where the transaction-oriented analyzers reside. Such an approach is employed by a number of schemes (Menasce and Muntz, 1978, Thomas, 1978 and Badal, 1979).

2) Forwarding of conflict-information

Although a conflict can be forwarded from a DM to an analyzer directly (Thomas, 1978, Stonebraker, 1978 and Dewitt and Wilkinson, 1980), an indirect forwarding may be advantageous. One such indirection is the hierarchical forwarding (Menasce and Muntz, 1978), which arranges all sites into a hierarchy. In forwarding the information on
conflicts upward in the hierarchy, the amount of information is compressed to reduce the communication load. Also, the detection of nonserializability is carried out at each site based on information gathered up to that point in the hierarchy. Another variation in indirection is chained forwarding (Menasce and Muntz, 1978 and Badal, 1979), where the destination for information to be forwarded is dynamically determined by the transactions involved. In each variation, conflict-information received at an analyzer results in indirect conflicts which are then forwarded to the appropriate analyzers. However, the first approach is site oriented while the second approach is information oriented.

While the term "multi-analyzer" indicates the distributed use of more than one site, the final analysis needed for correction is determined by some single site which has integral knowledge of conflict. Thus, multi-analysis may be considered centrally oriented.

3) Precedence selection This component is addressed in other literatures as the priority scheme. When nonserializability is detected (usually in the form of a cycle in the conflict-graph), one or more transactions have to be rolled back. Usually, it is desirable to roll-back a less important transaction. Such a transaction is said to have a lower precedence. The precedence of a transaction
can either be statically determined at the time the transaction enters the system (static precedence) or it can be dynamically determined when the need for a roll-back arises (dynamic precedence).

A very simple scheme of static precedence is to roll back every transaction involved in the nonserializability. In a sense, this concept does not require any precedence among transactions. This concept of total rejection may be advantageous in a small system or in a system with a very low level of conflict among transactions. Another static precedence selection is to create a total ordering among transactions based on the age of the transactions. The most widely used is the distributed timestamp scheme by Lamport (1978), although a simpler method such as the circulating sequence number or ticket (Le Lann, 1978) can also be employed.

Dynamic precedence selection is typically based on usages of resources or remaining time-to-completion information (Menasce and Muntz, 1978). Stonebraker (1978), however, simply chooses an arbitrary transaction to roll-back. A complex dynamic scheme is the one based on majority consensus. In such a scheme, the actual orderings between conflicting actions of two transactions at various sites will determine which transaction has higher precedence. The transaction which has the majority of orderings against it
(i.e., the majority dictates that it follows the other transaction) is rolled back. There are two forms of majority consensus. In a fully redundant database, the conflict-information appears at all sites. Thus, the majority in this case can be based on the number of sites (Thomas, 1978). In a partially redundant database, the two transactions do not have to meet at every site. In this case, the consensus is taken based on the number of sites where the actions of the two transactions meet.

b. Fractional conflict-analysis Fractional conflict-analysis is by definition distributed. In a fractional conflict-analysis, each DM acts as an analyzer. Each analyzer has only partial information on conflicts, usually those local to the analyzer's site. There are two components of this category: the dimensionality of conflict and the degree of redundancy.

1) Dimensionality of conflict This concept deals with the number of versions of a datum maintained as a history (Reed, 1983). Usually, only a single version of each datum is kept (not counting back-up copies for system recovery). The version kept reflects the latest change in the value of the datum by some write action. In a monoversion system, the history may be considered abbreviated to a temporal instant moving along the time axis of the datum's life. In contrast, a multiversion system
conceptually retains the complete history of a datum from the time of its first definition to the latest change by some write action. In practice, only a finite history is maintained. Reed represents this history by an ordered set of accesses to each datum. An update by some write action creates a new version which is added to the history of the target datum. A version occupies the interval on the time axis of the datum beginning at the version's creation and ending at the latest reference (by some read action) to that version.

If monoversion data is used, then any two actions which concurrently access the same datum actually access the latest and only available version of the datum. In order to guarantee serializability, an analyzer uses a pessimistic approach and rejects one of the conflicting transactions. In contrast, a pair of actions accessing a multiversion datum specify their version(s) by a pseudo-time instant. This is usually done by a system of distributed timestamps. Thus, a conflict means the coincidence both in space (the datum) and in time (the instant of access in the datum's life). A read action can always find a latest version earlier to the instant of access, and thus a read action is never rejected. On the other hand, a write action may intend to create a version where some version already exists. This would undo the history and cannot be allowed.
In such a case, the write action is rejected.

Since there is no exchanging of messages among analyzers, there is no information for dynamically establishing the precedence. Thus, a fractional analysis always requires a static selection where the precedence of a transaction is established when the transaction enters the system.

2) **Degree of data redundancy**

There are two possible subcategories, full redundancy and partial redundancy. In fractional conflict-analysis with fully redundant database, the actions of two conflicting transactions meet at every site. Thus, the transaction with lower precedence must always be rejected, since the conflicts are always resolved in the same way at every site. It is possible for an action of a lower precedence transaction to arrive at some site prior to a conflicting action of some higher precedence transaction. In this case, the site which has voted in favor of the first action has to reverse its vote in favor of the second action. Such a scheme is called **unanimous consensus with reversible vote** (Ellis, 1977 and Rahimi and Franta, 1979). Fractional conflict-analysis with a partially redundant database can employ unanimous consensus with reversible vote in a manner similar to the fully redundant case where a transaction with lower precedence is always rejected. This is implemented in
the WOUND-WAIT scheme by Rosenkrantz et al. (1978). In a partially redundant database, however, two conflicting transactions do not have to meet at every site, although they have to meet at some site (else they would not be in conflict). Suppose that two conflicting transactions require accesses at only one common site. Furthermore, suppose the transaction with the lower precedence has result in accesses at far more sites than the one with the higher precedence. If reversible unanimous consensus is used, the transaction with lower precedence will be rejected, creating a greater loss of resource usage. The remedy is to have the action of the transaction with higher precedence wait (deferred voting) if it arrives at the site after the one with lower precedence. In this case, if the transaction with lower precedence happens to access all required sites before any transaction with higher precedence, that transaction can pre-empt all conflicting transactions with higher precedence. Such a technique is termed unanimous consensus with nonreversible vote, and is implemented in the WAIT-DIE system by Rosenkrantz et al. (1978).

2. Waiting constraint

This component of the correction category differentiates between locking and nonlocking schemes. Locking is a specific technique used to avoid impending nonserializability and is subject to deadlock. Although
there are many forms of locking (Gray et al., 1975), the
technique often employed in distributed databases is two-
phased (Eswaran et al., 1976, Rosenkrantz et al., 1978 and
Menasce and Muntz, 1978), which maintains the logical
indisvisibility of a transaction during its life-time.
Integral conflict-analysis, in combination with a locking
technique, is commonly termed deadlock detection. This is
because integral conflict-analysis is performed on a
complete, disjointed portion of a conflict-graph (wait-for
graph) and nonserialzability is manifested by the presence
of a cycle in the graph which indicates those transactions
which are deadlocked. On the other hand, a locking
technique in combination with fractional conflict-analysis
is called deadlock avoidance. This is because fractional
conflict-analysis involves only some portion (with possibly
incomplete information) of the entire conflict-graph and a
pessimistic decision to roll-back has to be made if conflict
occurs.

3. Provision for local concurrency

At one extreme where concurrency is not locally
provided, the local database is considered logically
indivisible. Thus, concurrency at each DM depends on the
granularity of the local database and the parallel
capabilities of the local architecture.
4. **Categorization tree for correction**

The categorization of correction presented in the previous three sections is tabulated in Table 5.2. The labeling scheme is similar to that used for prevention in Table 5.1 and will be explained in the next section.
TABLE 5.2. Subcategories for Correction

<table>
<thead>
<tr>
<th>B: Correction</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Integrity of conflict-analysis</td>
</tr>
<tr>
<td>• A: Integral conflict-analysis</td>
</tr>
<tr>
<td>• Multiplicity of analyzer</td>
</tr>
<tr>
<td>• A: Mono-analyzer (system-oriented)</td>
</tr>
<tr>
<td>• B: Multi-analyzer (transaction-oriented)</td>
</tr>
<tr>
<td>• Forwarding of conflict-information</td>
</tr>
<tr>
<td>• A: Direct forwarding</td>
</tr>
<tr>
<td>• B: Indirect forwarding</td>
</tr>
<tr>
<td>• A: Hierarchical forwarding</td>
</tr>
<tr>
<td>• B: Chained forwarding</td>
</tr>
<tr>
<td>• Precedence selection</td>
</tr>
<tr>
<td>• A: Static</td>
</tr>
<tr>
<td>• A: Total rejection (no priority)</td>
</tr>
<tr>
<td>• B: Total ordering</td>
</tr>
<tr>
<td>• B: Dynamic</td>
</tr>
<tr>
<td>• A: Resource-usage oriented</td>
</tr>
<tr>
<td>• B: Time-to-completion oriented</td>
</tr>
<tr>
<td>• C: Arbitrary</td>
</tr>
<tr>
<td>• D: Majority consensus</td>
</tr>
<tr>
<td>• A: Full redundancy</td>
</tr>
<tr>
<td>• B: Partial redundancy</td>
</tr>
<tr>
<td>• B: Fractional conflict-analysis</td>
</tr>
<tr>
<td>• Dimensionality of conflict</td>
</tr>
<tr>
<td>• A: Monoversion data</td>
</tr>
<tr>
<td>• B: Multiversion data</td>
</tr>
<tr>
<td>• Degree of data redundancy</td>
</tr>
<tr>
<td>• A: Full redundancy</td>
</tr>
<tr>
<td>(unanimous consensus with reversible vote)</td>
</tr>
<tr>
<td>• B: Partial redundancy</td>
</tr>
<tr>
<td>• A: Unanimous consensus with reversible vote</td>
</tr>
<tr>
<td>• B: Unanimous consensus with nonreversible vote</td>
</tr>
</tbody>
</table>

| • Waiting requirement |
|   • A: Locking scheme |
|   • B: Nonlocking scheme |
| • Provision for local concurrency |
|   • A: Indivisible local database |
|   • B: Granular local database |

^aThe labeling scheme for components and subcategories is presented in Section E.
E. Categorization of Some Existing Concurrency Control Algorithms

The categorization scheme, developed in the main body of this chapter, is used to describe the relationship among concurrency control algorithms which were studied in this research. In order to develop a compact notation which clearly shows the categorization of an algorithm, an identification of an algorithm must obey the following syntactic rule.

\[
\begin{align*}
<\text{name}> & ::= <\text{alpha}><\text{name}> | <\text{alpha}><\text{component}> | <\text{alpha}> \\
<\text{component}> & ::= ( <\text{name}> )<\text{component}> | (<\text{name}> ) \\
<\text{alpha}> & ::= A | B | \ldots | Z
\end{align*}
\]

This identification corresponds to that used in Table 5.1 and 5.2. The subcategories of a given component are specified with the letters of the English alphabet, starting with the letter "A".

The syntactic rule implies that a category or subcategory may consist of one or more component. If there are two or more components, they are presented by a series of parenthesized descriptions. If there is only one component, it does not have to be parenthesized. As an example, suppose a prevention algorithm is identified by the name \(A(B(AA)(A))(A)\). Figure 5.1 shows the left-to-right parsing for this name (expression). Reading the end result of the parsing, the first A indicates the prevention...
category. Figure 5.2 illustrates this interpretation based on Table 5.1. The first component for prevention, the preanalyzer, is simplified and is centralized with a fixed central site and a total ordering describing its provision for concurrency. The other component, the enforcer, for prevention is OAS with consecutive timestamps.

<name> -> <alpha><component>
  -> A<component>
  -> A(<name>)<component>
  -> A(<alpha><component>)<component>
  -> A(B<component>)<component>
  -> A(B(<name>)<component>)<component>
  -> A(B(<alpha><name>)<component>)<component>
  -> A(B(A<name>)<component>)<component>
  -> A(B(A<alpha>)<component>)<component>
  -> A(B(AA)<component>)<component>
  -> A(B(AA)(<name>))<component>
  -> A(B(AA)(<alpha>))<component>
  -> A(B(AA)(A))<component>
  -> A(B(AA)(A))(<name>)
  -> A(B(AA)(A))(<alpha>)
  -> A(B(AA)(A))(A)

FIGURE 5.1. Left-to-right parsing for a name of an algorithm

In the following descriptions, the reference, itself, serves to name the algorithm. This is followed by a category identification along with a brief verbal description.
Prevention
Simplified preanalyzer
Centralized organization
Fixed central site
Total ordering in provision for concurrency
Enforcer with consecutive timestamps

A(B(AA)(A))(A)

FIGURE 5.2. Interpretation of an example of category names

Reference: Ellis (1977)
Category: B(B(A))(A)(B)(B)
Description: The fractional conflict-analysis of this correction algorithm is based on a fully redundant database with monoversion data and uses unanimous consensus with reversible vote. The algorithm is nonlocking with granularity of the local database for concurrency.

Reference: Badal and Popek (1978)
Description: The simplified preanalyzer for this prevention algorithm is distributed with nonconsecutive timestamps. Provision for concurrency is provided by the enforcer, but not by the preanalyzer. The OAS, which is based on nonconsecutive timestamps, does not reject any read action and employs strict ordering of write action processing with no reservation of write actions. The complexity of the algorithm lies in the reduction of messages by forwarding write actions only when requested by some conflicting read actions.
Reference: Bernstein et al. (1978), Bernstein et al. (1980a), Bernstein et al. (1980b) and Rothnie et al. (1980)
Category: A(B(B)(BBA))(B(B)(B)(A)(A))
Description: The simplified preanalyzer for this prevention algorithm is distributed with nonconsecutive timestamps. Provision for concurrency is provided by the preanalyzer through transaction aggregation static conflict-analysis which is translated into a distributed protocol at each of the DMs. The enforcer, which is passive and based on nonconsecutive timestamps, is quite complicated. This prevention algorithm allows rejection of read actions. The OAS also uses the TWR for the processing of write actions, although reservation of write actions is not provided.

Reference: Le Lann (1978)
Category: A(B(BA)(A))(A) and A(B(BA)(BA))(A)
Description: Both algorithms are preventive with a simplified preanalyzer. The preanalyzer is distributed and produces consecutive timestamps through the use of circulating tickets or counters. The first algorithm, the integrated distributed data sharing system, provides only a single stream of total ordering based on a single counter. The second algorithm, the partitioned distributed data sharing system, partitions the database and provides multiple streams of consecutive timestamps, each to a partition. The enforcer for both algorithms are simple because the timestamps are consecutive.

Reference: Menasce and Muntz (1978)
Category: B(A(A)(BA)(BA))(A)(B) and B(A(B)(BB)(BA))(A)(B)
Description: Both are correction algorithms based on a locking technique. The first algorithm, a hierarchical technique, employs the mono-integral-conflict analyzer with hierarchical forwarding of conflict information. The latter, a distributed algorithm, uses the multi-integral conflict-analyzer with chained forwarding of conflict information. Both algorithms, although labeled as using a dynamic precedence based on resource-usage, actually allow the use of any scheme of precedence selected by the system designer.
Reference: Rosenkrantz et al. (1978)
Category: B(B(A)(BA))(A)(B) and B(B(A)(BB))(A)(B)
Description: Both are correction algorithms with locking. Both employ a fractional conflict-analysis based on partial redundancy and monoversion data. The difference between them is that the former, the "wound-wait" method, uses a unanimous consensus with reversible vote, while the latter, the "wait-die" method uses a unanimous consensus with nonreversible vote.

Reference: Stonebraker (1978)
Description: This correction algorithm uses a mono-integral-conflict analysis (called SNOOP) on a granular local database. The forwarding of conflict information is hierarchical although only two levels exist in the hierarchy. Each site tries to detect and resolve local deadlock before forwarding the information to SNOOP. The precedence selection is simple: a transaction is arbitrarily selected for roll-back.

Reference: Thomas (1978)
Category: B(A(B)(A)(BDA))(B)(B)
Description: This correction scheme employs a multi-integral conflict analysis without locking. Provision for local concurrency is through the granularity of local database. Forwarding of conflict information is direct from each site (DM) to the analyzer (TM), in the case of broadcasting and otherwise from DM to DM in daisy-chaining. This algorithm is unique in that it uses a dynamic precedence selection with majority consensus. The voting is done on a fully redundant database and the majority is based on all participating DMs.

Reference: Badal (1979)
Category: B(A(B)(A)(AA))(B)(B)
Description: This correction algorithm employs a multi-integral conflict-analysis approach with direct forwarding of conflict information. The precedence selection, which resolves the nonserializability, is static with total rejection where all transactions are rolled back. The algorithm is nonlocking and granularity of local database provides concurrency.
Reference: Garcia-Molina (1979)
Category: A(B(AA)(A))(A) and A(B(AA)(BA))(A)
Description: There are four centralized preventive concurrency control methods embodied in this description. Each uses a simplified preanalysis with a fixed centralized site. The OAS is also simple, based on consecutive sequence numbers. Two algorithms (called CCA and CLA) belong to the first category. There is no provision for concurrency. The other two algorithms (called WCLA and MCLA) belong to the second category. They provide concurrency through centralized dynamic conflict analysis so that the enforcer can concurrently process nonconflicting transactions.

Reference: Herman and Verjus (1979)
Description: The simplified preanalyzer for this prevention algorithm is distributed with nonconsecutive timestamps. There is no explicit provision for concurrency either by the preanalyzer or by the enforcer, but each local database controller can supply the concurrency. The OAS does not allow rejection of read actions, TWR for write actions, or reservation of write actions.

Reference: Kaneko et al. (1979)
Category: B(B(A)(A))(B)(B)
Description: This correction algorithm employs a fractional conflict analysis on a fully redundant database with monoversion data. The voting scheme is unanimous consensus with reversible vote. The algorithm is nonlocking and employs local granularity for concurrency. This algorithm also employs a complex distributed clock which separates transactions into independent groups. Transactions only compete with others in the same group.

Reference: Rahimi and Franta (1979)
Category: B(B(A)(BA))(B)(B)
Description: This correction algorithm employs a fractional conflict analysis on a partially redundant database with monoversion data and uses unanimous consensus with reversible vote. The voting system is unique in that a DM does not independently reverse a vote but must ask the consent of the TM involved. This ensures that a single DM cannot undo the vote of the majority of DMs which improves the performance of the system at the cost of a higher communication requirement. The algorithm is nonlocking with local granularity for concurrency.
Reference: Seguin et al. (1979)  
Category: A(B(AB)(A))(A)  
Description: The simplified preanalysis in this prevention algorithm uses a centralized migrating control with a single stream of timestamps (no concurrency). The OAS is a simple one because of the consecutive timestamps. This algorithm employs a majority consensus on all sites for the two-phase commit to ensure reliability.

Reference: Chou and Liu (1980)  
Description: This prevention algorithm is similar to an earlier work (Herman and Verjus, 1979). The main difference is that this algorithm considers each local database as an indivisible entity. Provision for local concurrency is also possible through the local controller at each DM.

Category: B(A(A)(A)(BA))(B)(B)  
Description: The conflict analysis of this correction algorithm is integral with mono-analyzer. The forwarding of information is direct. Precedence selection is dynamic based on resource-usage. In this algorithm, when nonserializability occurs, the transaction that acquires the common datum last is rolled back. The algorithm is nonlocking with local granularity for concurrency.

Reference: Milenkovic (1980)  
Category: A(B(BB)(A))(B(A)(A)(B))(B) and B(B(A)(BA))(B)(B)  
Description: The first method (called P-protocol) is essentially the same as an earlier work (Chou and Liu, 1980). A major distinction is the use of reservation for write actions. The latter category (called Q-protocol) is a correction algorithm with fractional conflict-analysis on a partially redundant database with monoversion data and uses unanimous consensus with reversible vote. The algorithm is nonlocking with a granular local database.

Reference: Greene (1981)  
Category: A(B(AB)(A))(A)  
Description: The simplified preanalyzer for this prevention algorithm is centralized although migratable on a virtual ring based on a time-out mechanism. There are no provisions for concurrency in the preanalyzer although subqueues are employed at each DM. The OAS is simple because of the consecutive timestamps.
Category: B(B(A)(BA))(A)(B) and B(A(B)(BA)(AB))(B)(B)
Description: This correction algorithm contains two synchronization protocols. One of these handles interactions among transactions with both read and write actions. The other protocol governs the interactions between read-only transactions and transactions with both read and write actions. The former protocol employs fractional conflict-analysis and although it has not been explicitly stated, the algorithm can be used in a partially redundant database. The unanimous consensus is based on reversible votes. This protocol uses a locking scheme based on the 2PL technique. As for the latter protocol, a read-only transaction does not lock data. Control is based on multi-integral conflict-analysis which uses a two-level hierarchical indirect forwarding of conflict-information. The precedence scheme is static with rejection of only read-only transactions. This may be considered a form of total ordering between the two classes of transactions. The local database is granular.

Reference: Reed (1983)
Category: B(B(B)(BB))(B)(B)
Description: This correction algorithm employs fractional conflict-analysis with multiversion data. Although the algorithm does not specify the degree of data redundancy, it appears applicable in a partially redundant database using unanimous consensus with nonreversible vote. Since multiversion data is allowed, the algorithm is nonlocking with granular local database.

F. Summary

Two principal attributes were employed in developing the categorization scheme. The first is based on the two defining properties of the class CPSR. The second is the logical approach to concurrency control through prevention and correction. Each of the concurrency control algorithms studied in this research has features which realize these principal attributes. The categorization scheme was systematically developed, and each algorithm can be seen to
correspond to some combination of subcategories for either prevention or correction, as presented in Table 5.1 and 5.2. Some combinations of subcategories would, perhaps, not be feasible due to inherent inconsistency in philosophy or efficiency.

Although the list of categorizable algorithms presented here cannot be claimed to be complete, it seems a good representation of the attempts to control concurrent transaction processing in distributed database. The important concerns here are that the categorization of each algorithm is clear and simple, and similarities and differences among algorithms can be easily discerned.
VI. SYNTHESIS OF CONCURRENCE CONTROL ALGORITHMS

This chapter addresses an interesting issue which, although not directly related to the development of the categorization scheme presented in the previous chapter, is also aimed at clarifying an aspect of the turbulent situation in the field of concurrency control in distributed databases.

In particular, this section deals with an orderly synthesis of concurrency control algorithms. From a given mathematical description of a class of execution sequences, this synthesis produces abstract algorithms. Each resulting algorithm deals only with the problem of concurrency control, and is independent of the underlying physical database architecture. Three classes of execution sequences have been selected to demonstrate the synthesizing process. The classes and the reason for their selection are given in Section A. Section B presents the synthesis of corresponding centralized concurrency control algorithms, while Section C addresses the synthesis of distributed algorithms.

The synthesis process suggests the possibility of a systematic construction of an abstract concurrency control algorithm if the mathematical constraints of a class of execution sequences can be identified. However, it does not suggest the systematic derivation of mathematical
constraints. Also, since the three classes of execution sequences, are successive subsets of each other, the resulting algorithm for a smaller class can be based on constraints placed on top of those for the containing class.

A. Selection of Classes

Since the mathematical constraints are known for the classes depicted in Figure 4.3, they are obvious candidates to demonstrate the synthesis process. The three selected subclasses of the serializable class (SR) are

- conflict-preserving-serializable (CPSR) class,
- Q class, and
- two-phase-lock (2PL) class.

In this section, reasons for selecting these classes and rejecting others are given.

1. **CPSR class**

   The reason for selecting this class is simply that the defining properties of this class are a basis for the categorization of concurrency control algorithms. Chapter IV and V give two theoretical and one pragmatic reasons, respectively, of why this is so.

2. **Class Q**

   This class has been selected because a concurrency control algorithm which produces execution sequences in Q
can be implemented by further constraining the defining properties of the class CPSR. The abstract algorithms based on this class, and also that on the class 2PL, indicate the plausibility of an orderly synthesis of concurrency control algorithms.

3. Class 2PL

An execution sequence is in the class 2PL if and only if all of its transactions are two-phased. A transaction is two-phased if and only if it does not lock any more datum after it has unlocked some datum. Although there are a number of concurrency control algorithms which have been implemented specifically to produce the class 2PL, they are usually treated as locking algorithms. The algorithms synthesized in this chapter produce this class by a nonlocking technique similar to that of the class Q. This is important in the integration of the concepts of locking and timestamping.

4. Other classes

The remaining subclasses of SR, as depicted in Figure 4.3, are not considered. It follows from Theorems 4.1 and 4.3 that only three subclasses are actually excluded: P3, SDD-1 and SSR. The algorithm producing the classes P3 and SDD-1 is extensively documented (Bernstein et al., 1980a, Bernstein et al., 1980b, Rothnie et al., 1980), and requires
no attention in this chapter. The class SSR is not considered due to the possibility of NP-completeness in recognizing members of the class.

B. Synthesis of Centralized Concurrency Control Algorithms

In this section, the synthesis of three algorithms which produce the three corresponding selected classes of execution sequences is presented. The three algorithms are designed for a centralized database.

In synthesizing each algorithm, the mathematical constraints of the desired class of execution sequences are identified first. Then, two algorithms based on prevention and correction, respectively, are synthesized. An algorithm is considered a generator if it is based on prevention, or a recognizer if it is based on correction.

In contrast to the rest of this research, the synthesis presented in this section allows complex transactions (see Chapter III for a discussion of complex transactions). This is to show that algorithms which handle complex transactions need not be overly complicated. (This claim is also stated by Bernstein et al., 1979 and Papadimitriou, 1979.) Another aim of using complex transactions is to demonstrate a new graphical tool, called the three-level precedence graph. This tool has advantages over earlier graphical tools in that it can better handle the complexity of algorithms which
allows complex transactions.

1. Three-level precedence graph

The three-level precedence graph (3PG) is used to graphically describe centralized concurrency control algorithms which produce the three selected classes: CPSR, Q and 2PL. It is an integration of the concepts of a conflict-graph and a precedence relation used by other authors (Bernstein et al., 1979 and Badal, 1979). First, the disadvantages of a conflict-graph (CG) and a precedence relation are discussed. Then, 3PG is described.

a. Shortcomings of a conflict-graph and a precedence relation

The concept of a conflict-graph (Bernstein et al., 1979), which has been reviewed and discussed in Chapter IV, has special characteristics.

- The graph handles only simple dual-stepped transactions.
- The graph is based only on conflict among transactions.

The first characteristic implies that conflicts between any two simple dual-stepped transactions can be represented by at most three edges as shown in Figure 6.1. In a complex transaction, more than one action of the same type (read or write) may involve the same datum. Figure 6.2 shows the conflict-graph in such a situation. It is clear that the complexity of the conflict-graph increases rapidly in going
from a simple dual-stepped transaction to a complex transaction. Thus, some means must be devised to abstract the information in a conflict-graph.

FIGURE 6.1. Conflicting edges in CG between two transactions

FIGURE 6.2. CG for complex transactions

The second characteristic of a conflict-graph is the
fact that it is undirected. Presence of a cycle in an undirected graph denotes potential nonserializability. However, for a given nonserializable execution sequence, the direction of the cycle imposed by the precedence of operations cannot be depicted by placing a direction on the corresponding involved arcs in the undirected graph, without indicating an unnatural ordering of operations within some transactions. For example, consider the following execution sequence:


Figure 5.3 shows the conflict-graph after a direction is imposed. It does not contain a directed cycle. However, Figure 6.4 shows the directed cycle when the conflict-graph is based on a transaction as a whole. \( T_1 \) must precede \( T_2 \), since \( R_1[X] \) precedes \( W_2[X] \) in ES. Also, \( T_2 \) must precede \( T_1 \), since \( W_2[X] \) precedes \( W_1[X] \) in ES. Thus, while a conflict-graph is adequate for the theoretical analysis of serializability, the use of the analysis in some practical systems may require a more elaborate mechanism to handle the complex transactions.

The precedence relation, as defined by Badal (1979) represents the partial ordering between transactions as reflected by the actual order of conflicting actions over common data. The problem here is that the relation is
always expressed in term of the transactions, and thus is not suitable for use in a detailed analysis of interaction among the individual database actions. Thus, Figure 6.4 represents a Badal precedence relation.

To alleviate the shortcomings of both the conflict-graph and the precedence graph, a system of directed graphs is proposed. It is based on various levels of interpretation of complex transactions. Some definitions are in order. A mono-traced transaction is a transaction in which there exists a temporal total ordering of the database actions in the transaction. A detailed transaction is a transaction in which each of the actions involves only one
datum. Such an action is termed a detailed action. Any transaction $T_i$ can be represented in a detailed form by replacing each action $A_{ij}[X_1,X_2,...,X_n]$, where $A_{ij}$ is the action $j$ of $T_i$, by a sequence $A_{ij}[X_1]A_{ij}[X_2]...A_{ij}[X_n]$. It follows that a detailed execution sequence consists of transactions which are detailed or are represented in detailed form. The synthesis in this section assumes monotraced transactions which either are detailed or are presented in detailed form.

A usage $U_i[X]$ of a transaction $T_i$ on a datum $X$ is the time interval from the instant immediately before the first action involving $X$ (designated $A_{if}[X]$) to the instant immediately after the last action involving $X$ (designated $A_{ii}[X]$). Two usages are said to be in conflict if both are usages from different transactions, both involve the same datum, and at least one of them contains a write action on the datum. It should be noted that two transactions are said to be in conflict if they contain a pair of conflicting usages.

b. Definition of the 3PG
A three-level precedence graph (3PG) is a family of graphs which describes the interaction of the transactions in an execution sequence. The family consists of three types of graphs.

- First-level 3PG
- Second-level 3PG
• Third-level 3PG

Examples of the three types are shown in Figure 6.5, 6.6 and 6.7, respectively. The first-level 3PG is a directed graph in which a node represents a transaction and an edge represents the precedence between the conflicting transactions it connects. Thus, in Figure 6.5, the precedence relation is

T₃ < T₁ < T₂

where T₁ < T₃ stands for T₁ precedes T₃.

![Diagram of first-level 3PG]

FIGURE 6.5. An example of the first-level 3PG

The second-level 3PG is a partially directed graph. It consists of a matrix of nodes. Each node represents a usage of a datum. Each column of nodes represents a transaction. A vertical edge, connecting nodes in the same column, is undirected and serves only as visual aid to depict a transaction. The ordering of nodes in a column is arbitrary and does not signify any ordering of usages in the corresponding transaction. Each pair of conflicting usages
FIGURE 6.6. An example of the second-level 3PG

FIGURE 6.7. An example of the third-level 3PG
is connected by a directed edge called a conflict-edge. The
direction of a conflict-edge depends on constraints imposed
by any specific concurrency control algorithms. Thus, two
concurrency control algorithms may result in different set
of conflict-edges. An example of the second-level 3PG is
shown in Figure 6.6. A third-level 3PG is also a partially
directed graph which consists of a matrix of nodes. Each
node represents a detailed action on a datum. Nodes in the
same column represent actions in the same transaction, and
they are connected with an undirected edge. Each node is
labelled $R_{ij}$ or $W_{ij}$. The subscript $i$ denotes the
transaction number and $j$ denotes the $j$th-occurrence of an
action of this type in the transaction. The ordering of
nodes in a column, from top to bottom, represents the actual
temporal ordering, in ascending order, of actions in the
corresponding transaction. There are directed edges, each
connecting a pair of conflicting actions. The construction
of directed edges and their directions depends on individual
concurrency control algorithms. Figure 6.7 shows an example
of the third-level 3PG.

2. Synthesis of centralized concurrency control algorithms
producing the class CPSR

To review, an execution sequence is said to be in the
class conflict-preserving-serializable (CPSR) if there
exists an equivalent serial execution sequence in which the
order of conflicting actions is preserved.

The algorithms in this section are intended for use in a centralized database. However, some of the ideas are useful for special cases of distributed databases. For example, in the preventive algorithms, a major part of each algorithm could be used as a centralized preanalyzer. The enforcer, however, would require a much more elaborate algorithm than that required by a centralized database. To simplify the issue, the algorithm are proposed only for total centralized system. Distributed systems are discussed in Section C.

In the situation involving complex transactions, the definition of CPSR states that

- there is a partial ordering of the transactions involved, and
- for every pair of conflicting usages, the usages are ordered sequentially in the execution sequence, and the ordering corresponds to the precedence relation of the transactions in the partial ordering.

These two defining constraints are used in the construction of the 3PG for both the preventive and the corrective algorithms.

a. **Centralized preventive CPSR algorithm** Given a set of transactions. The following algorithm constructs
execution sequences in CPSR based on a preventive approach.

**Algorithm 6.1.**

1. A partial ordering is selected for the given set of transactions. The selection is based on some prearranged criteria of priorities, such as a timestamp scheme. The partial ordering is done on batches of transactions. Incoming transactions are grouped into batches and a partial ordering is constructed for each batch based on transactions in the batch. Batches are ordered sequentially. The first-level 3PG for each batch of transactions is constructed with the directions of edges conforming to the partial ordering.

2. The second-level 3PG is constructed as follows.
   - For each transaction in the graph (represented as a column of nodes), a node representing a usage on a datum $X$ is included if there is at least one action, of the transaction, that involves $X$.
   - An edge is drawn connecting each pair of conflicting usages.
   - A direction is given for each conflict-edge between usages of $T_i$ and $T_j$. This direction is the same as the direction of the edge
joining $T_i$ and $T_j$ in the transitive closure of the first-level 3PG.

3. The third-level 3PG is constructed from the second-level graph as follows. For each directed edge $(U_i[X], U_j[X])$ in the second-level graph, a directed edge $(A_{i1}[X], A_{jf}[X])$ is added in the third-level graph. It should be noted here that the numbers of directed edges in the second- and third-level graphs are equal. This is only true for algorithms producing the CPSR class. For algorithms producing other classes, there are extra edges due to the additional constraints for those classes.

4. The third-level 3PG is simplified in order to eliminate redundant ordering of actions. The result is a simplified third-level 3PG. The simplification is carried out by deleting any edge $(A_{i1}[X_m], A_{jf}[X_{m'}])$ if there is an edge $(A_{i1}[X_n], A_{kf}[X_{n'}])$ such that $A_{i1}[X_m]$ does not follow $A_{i1}[X_n]$, and $A_{kf}[X_{n'}]$ does not follow $A_{jf}[X_m]$. This mechanism is exemplified in Figure 6.8 and 6.9.

5. The directed edges in the simplified third-level 3PG act as the minimal precedence relations among actions. Any execution sequence constructed from
FIGURE 6.8. An example of the simplification of a third-level 3PG such a graph is in the class CPSR.

As an example, consider the following three mono-traced detailed transactions

\begin{align*}
& \quad W_{14}[X]W_{15}[W]W_{16}[Z] \\
\end{align*}

The construction of the three levels of 3PG is shown in
Notes: $A_{i1}[X_m]$ and $A_{i1}[X_n]$ are both equal to $W_3[X]$

$W_2[X]$ is $A_{jf}[Xm]$

$W_1[X]$ is $A_{kf}[Xn]$

FIGURE 6.9. Another example of the simplification of a third-level 3PG

Figures 6.5, 6.6 and 6.7, respectively. The simplified third-level 3PG is shown in Figure 6.10. For a centralized preventive CPSR algorithm, any execution sequence which does not violate constraints of the generated third-level 3PG is guaranteed to be in CPSR.

b. Centralized corrective CPSR algorithm

Given an execution sequence, the following algorithm determines whether it is in the class CPSR or not. In a corrective algorithm, only the first-level and the second-level 3PG are required to determine the membership of an execution sequence in the class CPSR. The algorithm is as follows.

Algorithm 6.2.
1. Given an execution sequence produced by a system of transactions, a second-level 3PG is constructed from the execution sequence using the following steps.

- The column of usages for each transaction is constructed by scanning the execution sequence once. Any usage which contains only read actions is marked.
- For any two successive actions of a datum X, \( A_i[X] \) and \( A_j[X] \), where \( i \) is not equal to \( j \), a directed edge \((U_i[X], U_j[X])\) is added in the
second-level 3PG, unless both usages contain only read actions.

2. The first-level 3PG is constructed by collapsing the second-level graph. If there is a directed arc from some usage in $T_i$ to some usage in $T_j$, then a corresponding arc must appear in the first-level corresponding 3PG.

3. If no directed cycle is found in the first-level graph, the execution sequence is in the CPSR class.

For example, consider the following execution sequence, constructed from Figure 6.10,

$$W_{23}[Y]R_{23}[X]$$

This execution sequence results in the second-level 3PG of Figure 6.6. The corresponding first-level 3PG is in Figure 6.5. Since it contains no cycle, $\text{ES}_3$ is in CPSR.

For an example of an execution sequence not in CPSR, consider the following execution sequence


This execution sequence results in the second-level graph in
Figure 6.11 which contains directed cycle as shown in Figure 6.12.

![Diagram of directed cycle](image)

**FIGURE 6.11.** Second-level 3PG for ES^4

![Diagram of first-level 3PG](image)

**FIGURE 6.12.** First-level 3PG for ES^4, showing a directed cycle

3. **Synthesis of centralized concurrency control algorithms producing the class Q**

From the definitions in Chapter IV, it is intuitive that the class CPSR is larger than the class Q. In the case of the class CPSR, the partial ordering of transactions must agree with the ordering of conflicting actions, while in the case of Q there are additional constraints imposed on the ordering between indirectly conflicting transactions. Two transactions T_i and T_j in an execution sequence are in indirect conflict with each other if both of the following conditions hold.
• There is at least one path connecting $T_i$ and $T_j$ in the first-level precedence graph, and
  • every such path must be of length greater than 1.
It follows that if any of the above paths is of length 1, then the two transactions are in **direct** conflict. Recall that in the case of the class $Q$, each transaction contains a serializability point inside its temporal boundary. A transaction is considered as being executed indivisibly at the time instant of the serializability point of that transaction (Papadimitriou, 1979). Two transactions are said to be ordered in a certain way if and only if their respective serializability points are also ordered in the same way in a given execution sequence. Thus, if a transaction $T_i$ precedes another transaction $T_j$ in the partial ordering, $T_i$ cannot follow $T_j$ in any execution sequence (i.e., $A_{i_f}$ cannot follow $A_{j_1}$). This is not the case with CPSR since there is no constraint on indirectly conflicting transactions. For example, consider the following execution sequence


which is in CPSR but not in $Q$. $T_1$ conflicts with $T_2$ through $X$. Since $R_{i1} < W_{21}$, $T_1 < T_2$. $T_1$ conflicts with $T_3$ through $Y$. Since $R_{31} < W_{11}$, $T_3 < T_1$. The ordering of this set of transactions is $T_3 < T_1 < T_2$. Thus, $T_3$ indirectly conflicts
with $T_2$ through $T_1$, and for $ES_5$ to be in $Q$, the serializability point of $T_3$ must precede that of $T_2$. But in $ES_5$, $T_2$ is actually executed before $T_3$. The reason inconsistency does not result is that $T_2$ and $T_3$ do not directly conflict with each other and the partial ordering is reflected through $T_1$.

The modification of the definition of class $Q$ to include complex transactions is as follows. A detailed execution sequence $ES$ is said to be in $Q$ if and only if, for each transaction $T_i$ in $ES$, a serializability point $S_i$ can be added such that

- $A_{iI} < S_i < A_{iI}$, and
- if $U_i[X]$ precedes $U_j[X]$ and at least one of them contains a write action, then $S_i < S_j$.

It should be noted here that the class $Q$ is properly contained in the class CPSR (Bernstein et al., 1979). The extended definition of class $Q$ leads to three theorems which are important in synthesizing centralized preventive and corrective $Q$ algorithms.

**Theorem 6.1.** Let $T$ be the set of transactions of an execution sequence $ES$ in CPSR. If for all $T_i$ in $T$, $T_i$ conflicts directly with every other transaction $T_j$, also in $T$, then $ES$ is in $Q$.

**Proof.** Let $PG_1$ be the first-level precedence graph representing $ES$, and $T_i$ be any transaction in $T$. Let $P$ and
F be the sets of transactions which precede and follow Ti in the partial ordering of PG1, respectively, dictated by the properties of CPSR. We have to prove that it is possible to select a serializability point Si inside Ti such that

- for all serializability points Sp of Tp in P, Si follows Sp, and
- for all serializability points Sf of Tf in F, Si precedes Sf.

The proof is divided into three steps.

- For all Sp of Tp in P, we can select Si in Ti such that Si follows Sp.
- For all Sf of Tf in F, we can select Si in Ti such that Si precedes Sf.
- The two Si's thus selected can further be selected such that they coincide.

For the first step, there exists at least one directed edge (Ap[X],Ai[X]) in the third-level precedence graph between Ti and each Tp in P. Thus, in ES, there exists an interval (Af,Ai), or in other words, Ap cannot follow Ai. Let Tp1,Tp2,...,Tpm be the elements of P. Then there are intervals

(Apf,Ai),(Apf,Ai),...,Apmf,Ai

in ES. If Si is selected to lie inside the shortest of such intervals, then it is possible to have a common Si which
follows any other $S_p$ of $T_p$ in $P$. The second step of the
proof is similar to that of the first step. For the third
step, let $T_{P_i}$ be the unique transaction in $P$ such that $A_{P_i}$
follows the first action of every other transactions in $P$.
Also, let $T_{F_j}$ be the unique transaction in $F$ such that $A_{F_j}$
precedes the last action of every other transactions in $F$.

Since $T_{P_i}$ conflicts with $T_{F_j}$, then $A_{P_i}$ must not follow
$A_{F_j}$. By definition of $T_{P_i}$ and $T_{F_j}$, the interval
$(A_{P_i}, A_{F_j})$ lies inside every other such interval between
any pair of $T_p$ in $P$ and $T_f$ in $F$. Also, this interval
overlaps $T_i$. This is because $A_{P_i}$ must precede $A_{i1}$, and
$A_{F_j}$ must follow $A_{if}$. Hence, step three of the proof can be
satisfied.

**Theorem 6.2.** Let $PG_1$ be the first-level 3PG of the
execution sequence $ES$ in CPSR. $ES$ is in the class $Q$ if for
any $T_i$ and $T_j$ in $ES$, $i$ not equal to $j$,

1. there exists at least a path between $T_i$ and $T_j$ in
   $PG_1$, and

2. for any such pairs of $T_i$ and $T_j$ with no path
   between them of length less than 2, if $T_i < T_j$ in
   $PG_1$, then $A_{if}$ precedes $A_{j1}$ in $ES$.

**Proof.** Let $P$ be the set of directed paths in $PG_1$. Define
$P_i$ in $P$ as an ordered tuple of transactions
$(T_{i_1}, T_{i_2}, \ldots, T_{i_n})$ where $T_{i_1}$ and $T_{i_n}$ represent transactions
on $P_i$ with no predecessor and no successor, respectively.
Any two transactions, which do not lie on some common path, have no ordering between them. Such pairs of transactions do not have to be considered because the ordering of their respective serializability points is not important. For transactions on the same path \( P_i = (T_{i_1}, T_{i_2}, \ldots, T_{i_n}) \), by the constraints of this theorem and of the class CPSR, \( A_{i_1} < A_{i_2}, \ldots, A_{i_n} \), where \( i_1 = i_2, \ldots, i_n \). Thus, it is possible to situate the serializability points so that \( S_{i_1} < S_{i_2} < \ldots < S_{i_n} \).

**Theorem 6.3.** An execution sequence ES is in class Q if and only if for any two transactions \( T_i \) and \( T_j \) in ES with at least a path between them in the first-level 3PG and no path is of length less than 2, if \( T_i < T_j \) in the first-level 3PG, then \( A_i < A_j \) in ES.

**Proof.** For any two transactions of an execution sequence ES in CPSR, there exists, in the first-level 3PG, either

- no path between the two transactions,
- a path of length 1, or
- only paths of length greater than 1.

Let \( G \) be the first-level 3PG for ES. If there is at least a pair of transactions in ES with no directed path in the first-level 3PG, then the transactions must belong to different disjoint portions of \( G \). Thus, such transactions define a set of disjoint graphs \( G_1, G_2, \ldots, G_m \) in \( G \), each of which does not contain any pair of transactions with no path between them. Thus the projection of ES over the set of
transactions in any \( G_i \) in \( G \) is in \( Q \) by Theorem 6.2.

a. Centralized preventive \( Q \) algorithm From Theorem 6.3, the extra constraint for the class \( Q \) is to ensure that any two indirectly conflicting transactions \( T_i \) and \( T_j \), \( T_i < T_j \) in the first-level 3PG, must be executed in such a way that \( A_{if} \) never follows \( A_{j1} \). To allow for as much concurrency as possible in the execution sequence, the extra constraint is imposed in the form of a directed edge \((A_{if}, A_{j1})\) in the third-level 3PG. Thus, a preventive algorithm which produces the class \( Q \) is as follows.

Algorithm 6.3.

1. The first-level, second-level and third-level 3PGs are constructed as in Algorithm 6.1.
2. For any \( T_i \) and \( T_j \) which do not conflict but have a path joining them in the first-level graph, and \( T_i < T_j \), insert a directed edge \((A_{if}, A_{j1})\) in the third-level graph.
3. Simplification of the third-level graph and the use of the simplified graph are the same as that described in Algorithm 6.1.

For example, Figure 6.13 shows the third-level graph of the following set of transactions.

\[
T_2 : R_{21}[Y]W_{21}[Y]W_{22}[Z]
\]
For this preventive $Q$ algorithm, any execution sequence which does not violate the constraints of the third-level 3PG, such as that of Figure 6.13, is guaranteed to be in $Q$. One of the possible execution sequence is as follows.

$$R_{12}[Y] W_{12}[Y]W_{22}[Z]$$

$T_2 < T_1 < T_3$

--- Extra constraint for class $Q$

FIGURE 6.13. Third-level 3PG for a preventive $Q$ algorithm

b. Centralized corrective $Q$ algorithm The algorithm which recognizes membership of the class $Q$ is as follows.

Algorithm 6.4.

1. Check whether the execution sequence is in the
class CPSR using Algorithm 6.2. If it is not, then it is not in \( Q \).

2. For all \( T_i \) and \( T_j \), with path of length greater than 1 in the first-level 3PG and \( T_i < T_j \), if in the execution sequence, there is at least an action of \( T_i \) preceding an action of \( T_j \), then the execution sequence is in \( Q \).

4. **Synthesis of centralized concurrency control algorithm producing the class 2PL**

   In this section, the concept of indirect conflict is extended. It is a practical constraint in constructing the 3PG for any execution sequence in the class 2PL.

   If a transaction \( T_i \) is augmented with an imaginary lock-point \( L_i \) (as in W2PL of Chapter IV), then the locking interval \( LI_i[X] \) for any datum \( X \) is defined as follows.

   • If \( L_i < A_{if}[X] \), then \( LI_i[X] \) is the interval from the point immediately before \( L_i \) to the point immediately after \( A_{if}[X] \).
   • If \( A_{if}[X] < L_i < A_{il}[X] \), then \( LI_i[X] \) is the interval from the point immediately before \( A_{if}[X] \) to the point immediately after \( A_{il}[X] \).
   • If \( A_{il}[X] < L_i \), then \( LI_i[X] \) is the interval from the point immediately before \( A_{if}[X] \) to the point immediately after \( L_i \).
Two usages $U_i[X_m]$ and $U_j[X_n]$, $i$ not equal to $j$, and $X_m$ not the same as $X_n$, are said to be in indirect conflict if $T_i$ and $T_j$ indirectly conflict with each other and on a path $T_iT_{k_1}T_{k_2}...T_{k_m}T_j$ in the first-level 3PG, there exist two usages, $U_{k_1}[X_m]$ and $U_{k_m}[X_n]$.

Theorem 6.4. An execution sequence ES is in the class 2PL if and only if

- it is in CPSR, and

- for any two indirectly conflicting usages $U_i[X_m]$ and $U_j[X_n]$, if $T_i < T_j$ in the partial ordering, then $U_i[X_m]$ precedes $U_j[X_n]$ in ES.

Proof. If ES is in 2PL, it is trivially in CPSR because the class 2PL is properly contained in the class CPSR. For the remaining portion of the if-part, let $B_i[X]$ and $E_i[X]$ represent the beginning and the ending points of $U_i[X]$ in $T_i$, respectively. Let $T_iT_{k_1}T_{k_2}...T_{k_n}T_j$ be the path between $T_i$ and $T_j$ in the first-level 3PG. Let $T_{k_1}$ and $T_{k_2}$ contain $U_{k_1}[X_p]$ and $U_{k_2}[X_p]$, respectively (the two usages are in direct conflict). Because each transaction obeys the 2PL locking protocol, every usage in the same transaction must overlap with each other. By the CPSR constraints,

$E_i[X_m] < B_{k_1}[X_m]$ and thus $E_i[X_m] < E_{k_1}[X_p]$. Since $E_{k_1}[X_p] < B_{k_2}[X_p]$, then $E_i[X_m] < B_{k_2}[X_p]$. Extending this for each successive pair of transactions, it follows that $E_i[X_m] < B_j[X_n]$. Thus, $U_i[X_m]$ precedes $U_j[X_n]$. 


For the only if part, the ordering of indirectly conflicting usages in ES leaves a temporal interval between each successive pair of indirectly conflicting usages. If all the usages for any transaction do not already have a common overlap interval (in which a lock point can be placed) then some of the usages can be extended to create such a common overlap interval.

Since the construction of 3PG for preventive and corrective 2PL algorithms is quite similar to those which produce the class Q, the actual algorithms are not stated here.

C. Synthesis of Distributed Concurrency Control Algorithms

In this section, the synthesis of distributed concurrency control algorithms is presented. The synthesis is systematic in the following sense. First, it uses a simple timestamping scheme to order transactions. This satisfies the first defining property of the class CPSR. Second, a simple waiting queue mechanism is used to enforce the ordering of conflicting actions. This satisfies the second defining property of CPSR. Thus, starting from the mathematical constraints on a class of execution sequences, abstract algorithms can be synthesized.

The algorithm, presented in this section as an example, has the following attributes.
1. The algorithms are based on the basic distributed database (BDD) model presented in Chapter III.
2. The algorithms produce execution sequences which are in the classes CPSR, Q and 2PL, respectively.
3. The algorithms control concurrent execution preventively.
4. Each algorithm concentrates only on the aspect of serializability of the resulting execution sequences.

1. A distributed preventive CPSR algorithm

The algorithm to be presented in this section is a preventive one and is similar to that proposed by Milenkovic (1980). The major difference is not in the technique used but in the presentation. In Milenkovic's algorithm, the technique is coincidentally based on the defining properties of the class CPSR, but the algorithm is not synthesized systematically. As a consequence, the algorithm contains details of the underlying database system and is not easy to understand or prove. In contrast, the preventive CPSR algorithm presented here concentrates on the serializability concept and thus is easy to understand and is trivially correct.

The algorithm presented here requires an extension of the BDD model as follows.
a. **Sequencer** To create a partial ordering, the set of TMs is augmented with a distributed process termed a **sequencer**. The sequencer generates an infinite set of ascending sequence numbers. If each event in the system is identified with one of the sequence numbers, then there is a total ordering of all events in the system. Sequence numbers are called **timestamps**. Each timestamp is a unique number. Adjacent timestamps in the sequence do not have to be consecutive. TMs use these timestamps to order events in the system. The distributed timestamp technique devised by Lamport (1978) is currently the most widely used, but any other technique can also be employed. One such technique is the circulating ticket devised by Le Lann (1978). The technique circulates a counter among the TMs on a virtual ring, and thus the timestamps are simply consecutive integers.

b. **TM** The function of a transaction manager (TM) is basically as follows.

1. Upon receiving a transaction, the TM requests a new timestamp for the transaction from the sequencer.

2. The transaction is broken down into its component database actions, each of which also carries the timestamp of the transaction.

3. Each database action, either a read or a write
action, is forwarded immediately to its appropriate data manager (DM). For a read action, the action waits for its turn to read the database local to the target DM. For a write action, the action acts as a reservation at the target DM, so that the DM knows as soon as possible of the action's presence in the system. The values of the data involved in the write action will be sent later from the TM after they have been produced. Thus, a write action is processed by the DM when its turn comes up and its values have been received. (The scheduling of actions at a DM is described later on.)

4. After all data required by the transaction have been returned from DMs, the TM starts the computation phase of the transaction. The results obtained at the end of the computation phase are forwarded to appropriate DMs, where the write actions have already made reservations. The forwarding of these results also signifies the beginning of the second phase of the two-phase commit protocol.

5. The TM then informs the user of the completion of the transaction.
c. **DM** In this algorithm, a data manager (DM) executes actions received from TMs in such a way that the effect on the database state is equivalent to the execution of the actions in timestamp order. In its basic form, a DM collects actions in timestamp order into a *basic waiting queue*. The DM removes an action with the oldest timestamp from the waiting queue for processing. In order to guarantee that a selected action is the oldest and will not be superseded by some older action arriving late, the following two constraints are imposed.

1. Messages sent from a TM to a DM must be pipelined, i.e. they are received in the same order as sent, and furthermore they are in timestamp order.

2. The DM has received, from each TM in the system, at least an action with a timestamp later than that of the action to be considered oldest.

If the two constraints hold, then the action in question is the oldest at the DM at that time. These two constraints comprise the concept of *oldest-action-selection*.

d. **The complex waiting queue** The basic waiting queue described above requires actions at a DM to be executed in timestamp order, regardless of the data involved in the actions. The technique implies that nonconflicting actions at a DM have to wait in the queue for their turns,
and thus the technique of the basic waiting queue prevents concurrency among nonconflicting actions. The complex waiting queue allows for a more flexible ordering of actions to increase the performance of the DM.

In order to increase the performance of the waiting queue, the basic waiting queue is augmented with three features.

- Subqueues
- Batch of data dependencies
- Detailed action handling

1) **Subqueues** To allow for concurrency among nonconflicting actions, the waiting queue at each DM is subdivided into subqueues, each for a datum in the DMs database. Subqueues can be created dynamically on demand. Actions in a subqueue are ordered according to their timestamps. Thus, the concept of oldest actions applies to each subqueue, each of which involves a different datum from the others. Note that an action in the queue must be in detailed form, i.e., it involves only one datum. (The third component, to be described, will handle the actions not in detailed form.)

2) **Batch of data dependencies** If actions in a subqueue are processed in strict sequence, then the performance of the system may still be severely limited. In order to improve the performance inside a subqueue, this
algorithm employs the integration of two techniques, namely: the Thomas Write Rule (TWR) (Thomas, 1978) and the reservation of write action. The processing of actions in timestamp order at a DM implies that, for write actions, the state of the database local to the DM always reflects the last, or youngest, version of each datum. Thus, a write action can be discarded if it arrives at a DM with the timestamp earlier (older) than the timestamp of its datum in the database. This means that write actions do not have to be applied to the datum in timestamp order as long as an older write action does not supersede a younger one. The benefit of the other technique, the reservation of write action, can be demonstrated as follows. Consider a system with three TMs, with the following actions destined for a certain $DM_1$.

$$\begin{align*}
TM_1 & : W_1[X] \\
TM_2 & : R_2[X] \\
TM_3 & : W_3[X]
\end{align*}$$

Suppose $DM_1$ receives $W_1R_2$ in that order of timestamps and places them in the subqueue of the datum $X$. If the system did not employ the reservation of write action, $W_1$ would have the value of $X$ associated with it. Nevertheless, the writing of $W_1$ and the reading of $R_2$ could not be carried out because of the absence of $W_3$, which might have an earlier
timestamp than \( W_1 \) or \( R_2 \) or both. Thus, if \( T_3 \) is lengthy, \( T_1 \) and \( T_2 \) will be held up until the value of \( X \) for \( W_3 \) is available. However, using the reservation of write technique, \( W_3 \) will be sent immediately upon its inception without the value of \( X \). Suppose the reservation of \( W_3 \) changes the ordering in the subqueue to \( W_1R_2W_3 \). In this case, \( W_1 \) and \( R_2 \) can be processed immediately by \( DM_1 \). If the reservation of \( W_3 \) changes the ordering in the subqueue to \( W_1W_3R_2 \), then \( W_1 \) can proceed but \( R_2 \) has to wait for the value of \( X \) for \( W_3 \).

The integration of the above two techniques can lead to an additional improvement in performance. Consider the situation where \( DM_1 \) receives \( W_3 \) in such a way that the ordering in the subqueue of \( X \) becomes \( W_3W_1R_2 \). Since \( W_3 \) is a reservation, the value of \( X \) for the action is absent. Normally, the remaining actions cannot be processed. But if the TWR is applied at this point, it can be seen that \( W_1 \) can be processed before \( W_3 \) (if its corresponding value of \( X \) is present and \( DM_1 \) receives another action from \( TM_3 \) with timestamp later than \( R_2 \)). Also, since the value of \( X \) required by \( R_2 \) is that produced by \( W_1 \) because of the immediate dependency of data, then \( R_2 \) can also be processed without waiting for the value of \( X \) for \( W_3 \) (provided that \( DM_1 \) receives actions from \( TM_1 \) and \( TM_3 \) with later timestamps than \( R_2 \)). When the value of \( W_3 \) is received later on, it is
discarded because the value has been obsoleted by the processing of $W_1$. The concept of the integration of these two techniques is termed the batch of data dependency and its detailed algorithm is to be presented later on in this section.

An interesting consequence of not using the reservation technique is that if write actions are not reserved, then a TM must hold any write actions of the transaction it is processing until the computation phase is done. As a result, the pipelining constraint may force the TM to process transactions sequentially whether they are in conflict or not. Thus, the reservation of write actions may allow more concurrent processing of transactions at each TM.

3) Detailed action handling An action from a TM to a DM may involve more than one datum. Such an action complicates the data dependencies within subqueues because an action may depend on actions in more than one subqueue. To simplify this problem, each action received by a DM is broken down into several actions, each involving only one datum. Such actions are said to be in detailed form. The detailed actions are then placed in their respective subqueues. A set of flags is kept for the detailed actions belonging to the same original nondetailed action. Each detailed action goes through its subqueue in the normal fashion, and when it has been processed (because it becomes
oldest in its subqueue) the corresponding flag is set. When all flags signify that their corresponding detailed actions have been processed, the original nondetailed action is considered completed.

e. **Algorithm for the complex waiting queue**

The implementation of the complex waiting queue at each DM can be expressed by the following algorithm.

**Algorithm 6.5.**

1. When an action is received by a DM, it is broken down into its component detailed actions, each of which handles a datum of the original nondetailed action.

2. A set of complete flags is created for the nondetailed actions. Each flag is initially set to zero, which signifies that its corresponding detailed action has not yet been processed by the DM.

3. The detailed actions are inserted in timestamp order into the waiting queue of the DM.

4. Actions in the waiting queue are grouped into two sets. The **consecutive set** contains actions whose ordering by timestamp cannot be changed by subsequent arrivals of actions. The **nonconsecutive set**, on the other hand, may have its ordering of actions disturbed by subsequent
arrivals. A detailed action enters the waiting queue as a member of the nonconsecutive set. The action can migrate to the consecutive set when there exist at least one action with a later timestamp from each of the other TMs in the system.

5. The consecutive set is divided into subqueues, each to a datum.

6. A subqueue is divided into batches of data dependencies. A batch begins at a write action and includes subsequent read actions in ascending timestamp order up to but not including the next later write action. The value of the datum in the database acts as the write action for the first batch of the subqueue of that datum. In this case, a pseudo-write action is created to house the value. Note that a pseudo-write action is needed when a subqueue is created. This pseudo-write action may be subsequently replaced by a later "actual" write action. The last batch of a subqueue ends at the last read action of the subqueue. The last batch is considered incomplete because there may be further read actions incorporated into the subqueue.

7. A batch of data dependencies is ready when the
corresponding value of the write action of the batch has been received from the action's TM.
The DM processes a ready batch by assigning the value of the write action to the dependent read actions in the batch. All actions in the batch are then considered processed and their corresponding complete flags are set. The value of such a read action is added to the set of values in its parent nondetailed action. The read actions are then deleted from the waiting queue. When all read actions of a batch have been processed, and the batch is not the last one of the subqueue, the batch is deleted from the subqueue.

8. If there is only one batch (the last batch) in the subqueue, and the value of the write action has been received, then after the read actions have been processed the value of the write action of the batch is copied into the database and the subqueue is deleted.

9. When all complete flags of a nondetailed action are set, the action is considered completely processed. All its required values are then returned to the originating TM if it is a read action, or an acknowledgement is sent to inform
the TM of the completion of the write phase for
that DM if it is a write action.

f. An example of the complex waiting queue To
illustrate the mechanism of Algorithm 6.5, consider a
snapshot of the waiting queue at a data manager \( DM_i \) in
Figure 6.14. For simplicity, there are only four TMs in the
system and only three data at \( DM_i \). The dummy action is sent
periodically by a TM when it has nothing to do. Each dummy
action contains only the timestamp. The dummy action allows
the processing of the waiting queue to proceed without
having to wait for normal actions to arrive regularly from
the TMs.

The value of a write action in Figure 6.14 is either
"yes" or "no", signifying that the write action has or has
not received the value of its datum. A read action does not
require this designation of its value because the status of
the value is not involved in determining the status of the
batches. Actions in the waiting queue are assumed to be
ordered by their timestamps.

The consecutive set in Figure 6.14 consists of actions
1 through 11. The last action in the set, action 11, has
actions with later timestamps from all the other TMs. These
are actions 12, 13 and 14. Action 12 is in the
nonconsecutive set because \( DM_i \) has not received a later
action from \( TM_j \). It may turn out that the next action from
FIGURE 6.14. A snapshot of the waiting queue at \( \text{DM}_{1} \)

<table>
<thead>
<tr>
<th>Entry</th>
<th>Action-type</th>
<th>TM #</th>
<th>Datum</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>write</td>
<td>1</td>
<td>X</td>
<td>no</td>
</tr>
<tr>
<td>2</td>
<td>read</td>
<td>2</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>write</td>
<td>3</td>
<td>Y</td>
<td>no</td>
</tr>
<tr>
<td>4</td>
<td>read</td>
<td>4</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>read</td>
<td>1</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>write</td>
<td>2</td>
<td>Y</td>
<td>yes</td>
</tr>
<tr>
<td>7</td>
<td>write</td>
<td>3</td>
<td>X</td>
<td>no</td>
</tr>
<tr>
<td>8</td>
<td>write</td>
<td>4</td>
<td>X</td>
<td>yes</td>
</tr>
<tr>
<td>9</td>
<td>read</td>
<td>1</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>write</td>
<td>2</td>
<td>Z</td>
<td>no</td>
</tr>
<tr>
<td>11</td>
<td>read</td>
<td>3</td>
<td>Z</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>read</td>
<td>4</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>13</td>
<td>read</td>
<td>1</td>
<td>Z</td>
<td>-</td>
</tr>
<tr>
<td>14</td>
<td>write</td>
<td>2</td>
<td>X</td>
<td>no</td>
</tr>
<tr>
<td>15</td>
<td>dummy</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

\( \text{TM}_{3} \) has a timestamp older than action 12. In that case, the new action is to be the next one included in the consecutive set instead of action 12.

There are three subqueues in the consecutive set of Figure 6.14 for the data \( \text{X, Y and Z} \), respectively. This is shown in Figure 6.15. The subqueue for \( \text{X} \) consists of actions 1, 2, 4, 7, 8 and 9. In this subqueue, there are three batches of data dependency, namely \( (1,2,4), (7) \) and \( (8,9) \), each of which is started with a write action. The third batch, which is the last batch, is not complete...
because if there is an action on X arriving from TM₃ with timestamp later than action 12, then action 12 has to be included in this third batch. Nevertheless, the third batch of X is ready, and the value of X from action 8 can be given to action 9 immediately. If action 9 is part of some nondetailed action Aᵢ, then the value of action 9 is attached to Aᵢ, and the corresponding complete flag is set. Action 9 is then deleted from the third batch of this subqueue. Action 8 is not deleted, however, because this batch is the last batch.

Suppose now DM₁ receives the value of X for action 1. The first batch is ready and is processed. The batch is then deleted because the value of X for action 1 is superseded by that for action 7 and 8.

The second batch of the subqueue for X shown in Figure 6.15 is rather interesting. The batch has no read action at all, and thus the write action 7 is considered dead (its value is not used) local to DM₁. Thus, DM₁ can delete the second batch as soon as it is found to be trivial. DM₁ does not have to wait for the value of X of action 7 at all. Note that a last batch cannot be declared as trivial because its growth has not been completed. The life-time of a trivial batch depends on the design criteria. A DM may wait until the value of the write action in the trivial batch arrives, and then discard the batch, together with that
FIGURE 6.15. Subqueues of the consecutive set in Figure 6.14

write action or a DM may take a more drastic action. When a batch is found trivial, it can be discarded immediately, and the nondetailed write action, which originates the detailed write action of the batch, can be informed. If all detailed actions of a nondetailed write action belong to trivial batches, the write action is discarded, and a message is
sent to the originating TM informing the TM that the write action is a dead one. If the action is the only write action in the transaction, the TM may decide to abandon (i.e., commit without processing) the transaction altogether. This mechanism, when carried to its extreme, may result in withdrawals of the read actions of the dead transaction from various DMs, and can thus further result in more trivial batches. The extent of this elimination of dead transactions may have a profound effect on the performance of the system which contains heavily used data. On the other hand, the extra messages required may prohibit any significant application of such technique.

2. Distributed preventive $Q'$ algorithms

In this section, two distributed concurrency control algorithms are presented. The execution sequences produced by each are in a proper subset $Q'$ of the class $Q$. This restriction of the class $Q$ simplifies the corresponding algorithms in that synchronization among indirectly conflicting transactions is not necessary.

Any execution $ES$ is in class $Q'$ if and only if the following hold.

1. $ES$ is in CPSR, and
2. for any pair of directly conflicting transactions $T_i$ and $T_j$, $T_i < T_j$ in the ordering imposed by CPSR implies $A_{i,f} < A_{j,f}$ in $ES$. 
Of the two distributed preventive $Q'$ algorithms to be presented in this section, the former algorithm, the simple preventive $Q'$ algorithm, places restrictions on the database actions allowed in the system for simplicity in the synthesis process. The latter, the complex preventive $Q'$ algorithm, extends the first algorithm to represent a more realistic environment in a distributed database system. The presentation of these algorithms introduces two concepts pertaining to the synthesis of concurrency control algorithms in distributed databases. One of these concepts is the orderly extension of the procedure of synthesis of algorithms which produce execution sequences in the next containing class of execution sequences, namely the class CPSR. This is because $Q'$ is a subclass of CPSR and hence exhibits defining properties which are special cases or restrictions of that of the class CPSR. The other concept is the virtual ordering, which results from the presence of serializability points associated with individual transactions. This concept places additional restrictions on the resulting execution sequences of a preventive $Q'$ algorithm, and on other algorithms producing subclasses of $Q'$.

**Theorem 6.5** The class $Q'$ is a proper subset of $Q$.

**Proof** Let $T_i$ and $T_j$ directly conflict. Assume $T_k$ conflict indirectly with $T_i$ through $T_j$, and $T_j < T_k$. If
A_i f \prec A_j f$ and $A_j f \prec A_k f$, then $A_i f \prec A_k l$, and Theorem 6.3 is satisfied. Hence, $Q'$ is a subset of $Q$. $Q'$ is a proper subset of $Q$ because the execution sequence


is in $Q$ but not in $Q'$.

It should be noted here that for a system involving only simple dual-stepped transactions, the constraints between any two directly conflicting transactions will be $R_i \prec R_j$.

The definition of $Q'$ requires synchronization between directly conflicting transactions, which is relatively simple to implement because a pair of directly conflicting transactions conflict with each other through common DMs. In contrast, the synchronization for class $Q$ must include indirectly conflicting transactions as well as directly conflicting ones.

a. **Simple preventive $Q'$ (SPQ) algorithm** For simplicity, each database action in this algorithm involves only one data manager (DM). Furthermore, each action is a detailed one (i.e., it involves only one datum) during the life-time of its transaction. This implies that the database is nonredundant, and a transaction involves at most two data, one to be read and the other to be written. Nondetailed actions with redundant database are treated in the complex preventive $Q'$ algorithm.
The algorithm for SPQ is an extension of Algorithm 6.5. The modifications result in the following algorithm.

Algorithm 6.6. For each transaction $T_i$ which has its read action $R_i$ and write action $W_i$ present at $DM_p$ and $DM_q$, respectively:

1. When $W_i$ is the first entry in its appropriate subqueue, $DM_q$ sends a message "initiate $T_i$" to $DM_p$. ($DM_q$ knows of $DM_p$ by information associated with $W_i$).

2. $R_i$ is processed when
   - it is included in the appropriate subqueue,
   - the write action heading the batch of $R_i$ has received its corresponding value, and
   - $DM_p$ has received "initiate $T_i$" from $DM_q$.

b. Virtual ordering From Algorithm 6.6, it can be seen that there is no concurrent processing of batches of data dependency in the same subqueue. This is because, in addition to the physical orderings among conflicting actions of an CPSR algorithm, an SPQ algorithm requires a logical or virtual ordering among transactions based on the intuitive serializability points (Papadimitriou, 1979). Recall that in class Q, each transaction is assigned an imaginary serializability point bounded inside the physical boundary of the transaction, i.e., $A_i < S_i < A_i$, where $S_i$ is the serializability point of the transaction $T_i$. If a
transaction \( T_i \) precedes another transaction \( T_j \) in the partial ordering among transactions, then \( S_i \) must precede \( S_j \) in the resulting execution sequence. The synchronization technique based on class \( Q \) must allow for this ordering requirement. Thus, although a CPSR algorithm relies solely on the physical ordering of actions reflected through the local database state of each DM, the SPQ algorithm must adhere to the ordering of action processings in order that the global execution sequences will still be in \( Q \). In other words, the CPSR algorithm perceives the global execution sequence through the system's database state, while the \( Q \) algorithm perceives the global execution sequence through the actual course of action processing.

The concept of virtual ordering is applicable to algorithms producing the class \( Q \) and its subclasses. As a result, corresponding algorithms may be less efficient than those producing the class CPSR.

c. **Complex preventive \( Q' \) (CPQ) algorithm** This algorithm handles nondetailed database actions in a redundant distributed database. For simplicity, only simple dual-stepped transactions are still considered.

For any nondetailed transaction \( T_i \) with a read action \( R_i \), we define \( SR_i \) as the set of detailed-read actions, called read subactions of \( R_i \). \( SW_i \) is defined for \( W_i \) similarly. In the CPQ algorithm to be presented, a
transaction $T_i$ is said to be in progress if one or more $R_{i,k}$ in $SR_i$ has been processed by some DM. A read subaction $R_i'$, selected randomly from $SR_i$, is defined as the primary subaction of $T_i$. $R_i'$ has associated with it a set of ready flags, each for a write subaction in $SW_i$. When all these flags are set, $T_i$ is said to be enabled, and any of its read subactions can be processed by the DMs.

Algorithm 6.7 This algorithm is an extension of Algorithm 6.5. The modifications are as follow.

1. A write subaction is considered ready if
   • it is the first subaction of the subqueue, or
   • the transactions of all subactions in its immediately preceding batch have been in progress.

2. When a write subaction is ready, it informs its primary subaction.

3. A transaction goes into the enable state when its primary subaction has received the ready status corresponding to all of its write subactions. The primary subaction then informs every read subaction of the enable status.

4. A read subaction can be processed when
   • it is included in a subqueue of some DM,
   • the write subaction of its batch has received the value of the associated datum,
206

• its transaction has been enabled or has been in progress, and
• it is the first read subaction in the batch, or the read subaction which is its immediate predecessor in the batch has been processed.

5. A read subaction informs its primary subaction of the in-progress status if
• it has been processed, and
• its transaction has not already been in progress.

6. A transaction goes into an in-progress state when its primary subaction has received in-progress status from one or more of its read subactions. The primary subaction then informs all of its subactions of the in-progress status.

7. A write subaction can be processed when it has received the value of its datum.

3. Distributed preventive 2PL algorithms

By examining the characteristics of execution sequences in the class two-phase lock (2PL), a synchronization technique can be specified to enhance a preventive CPSR algorithm so that resulting execution sequences are in 2PL. This process is similar to that of preventive Q algorithms developed in the previous section, and hence supports the plausibility of systematically devising concurrency control
algorithms which produce various classes of execution sequences.

a. **Synchronization technique for preventive 2PL algorithm**

An execution sequence which is in $Q$ can violate the defining properties of the class 2PL. This can be shown by considering the following three transactions.

\[
\begin{align*}
T_1 &: R_1[X]W_1[Y] \\
T_2 &: R_2[Y]W_2[Z] \\
T_3 &: R_3[W]W_3[Z]
\end{align*}
\]

The execution sequence

\[
\]

is in $Q$, with $T_3 < T_2 < T_1$, because $T_3$ which indirectly conflicts with $T_1$ has $R_3 < W_1$ and thus satisfies the requirement of Theorem 5.3. $ES_1$, however, is not in 2PL. This is because the locking interval of $Y$ for $T_1$ must follow that of $T_2$, and the locking interval of $Z$ for $T_3$ must precede that of $T_2$. Since the locking intervals of $T_1$ must overlap to conform to the 2PL constraint, the locking interval of $Y$ for $T_1$ must follow the locking interval of $Z$ for $T_3$. This is not possible since $W_1[Y]$ precedes $W_3[Z]$ in $ES_1$. If $W_3$ were to precede $W_1$, then $ES_1$ would be in 2PL.

It can be seen that $W_3$ conflicts indirectly with $W_1$ through the actions of $T_2$, and since $T_3 < T_1$ in the partial ordering,
then $W_3$ should also precede $W_1$ in the execution sequence. This observation leads to Theorem 6.4.

b. Simple preventive 2PL (SP2PL) algorithm For a system employing only simple dual-stepped transactions, the extra synchronization to make an execution sequence in CPSR fall into 2PL is conceptually shown in Figure 6.16. It is assumed that the transactions involve only detailed actions and the database is nonredundant. This assumption is the same as in the SPQ algorithm. It can be seen that although the actions in sets A and C are indirectly in conflict through $T_i$, the transactions of these actions precede $T_i$ in the partial ordering. Hence, their actions do not have to be ordered in the execution sequence. Similar argument also applies to the sets B and D. For the sets A and D, actions in set A precede $R_i$. $R_i$ trivially precedes $W_i$ which in turn precedes action in set D. Thus, the actions of the two sets are properly ordered through the relationship between $R_i$ and $W_i$.

For the actions of the sets B and C, there is the possibility that an action in C is processed before an action in B, rendering the execution sequence not in 2PL. Hence, synchronization is required among actions of these two sets. Modifications of Algorithm 6.5 to accomplish this synchronization result in the following algorithm.

Algorithm 6.8.
Subqueue of X: \( A \ R_i \ B \)
Subqueue of Y: \( C \ W_i \ D \)

Notes:

1. A and B are the sets of write actions conflicting directly with \( R_i \), with their transactions preceding and following \( T_i \) in the partial ordering, respectively.

2. C and D are the sets of read and write actions conflicting directly with \( W_i \), with their transactions preceding and following \( T_i \) in the partial ordering, respectively.

Synchronization requirements:

1. No synchronization between A-C, A-D and B-D.

2. Synchronization is required between B and C.

FIGURE 6.16. Synchronization requirements for 2PL algorithm
1. A write action $W_i$ will send a message (actually sent by the DM) notifying its read action $R_i$ that the "potential" of indirect-conflict is gone when
- $W_i$ is included in some subqueue,
- all preceding write actions in the subqueue have been processed, and
- all preceding read actions in the subqueue have been notified of the absence of potential indirect conflict by their write actions.

2. A read action $R_i$ can be processed when
- it is included in some subqueue,
- it has received "potential indirect-conflict gone" message from $W_i$, and
- its batch is ready (the write action of the batch has received its value).

3. A write action $W_i$ can be processed as soon as it has received its value.

c. Complex preventive 2PL (CP2PL) algorithm   The CP2PL algorithm, which handles nondetailed actions and the redundant database, is based on Algorithm 6.8 with some modifications to handle the distribution of an action into its detailed and duplicated subactions. The mechanism is similar to that employed in the CPQ algorithm, and hence is not described here.
D. Summary

This chapter presented an interesting issue of synthesizing centralized and distributed concurrency control algorithms. The presentation demonstrated the feasibility of an orderly construction of abstract algorithms using mathematical properties of various classes of serializable execution sequences. Furthermore, since the algorithms produce successive containing classes, the resulting procedures of synthesis can be based progressively on the procedures for the next containing classes. Since the algorithms are abstract, they are easier to understand than actual descriptions of their implementations. Correctness of the algorithm can be proved prior to attempting the actual implementation.
VII. CONCLUSION

A. Research Summary

The problem of concurrency control in distributed databases has led to the development of various concurrency control protocols, a number of which were described in Chapter II. The framework on the problem of concurrency control was defined in Chapter III. This framework was represented by the BDD model which is an abstract model for a working distributed database. In Chapter IV, the concept of serializability as a sufficient condition for database consistency was discussed. The class of serializable execution sequences (SR) and its subclasses were investigated. An analysis of the subclasses yielded a distinguished class, the conflict-preserving serializable (CPSR) class, with two defining properties which served as a basis for the categorization scheme developed in Chapter V. Also in Chapter V, the concurrency control algorithms studied in this research were categorized according to the categorization scheme developed. The result is encouraging because the categorization scheme can account for all the protocols, the aggregation of which is a good representative in this field. Chapter VI presented an orderly synthesis of concurrency control algorithms, both centralized and distributed.
B. Discussion

The field of concurrency control in distributed databases is in a turbulent state due to a variety of solutions to the problem. Thus, it is important to establish an integrated and comprehensive categorization for all the protocols proposed. Although qualitative in nature, this research establishes the feasibility of such a categorization scheme based on the serializability concept which is widely approved. The categorization scheme developed provides an orderly way to view various concurrency control algorithms and their interrelationships. Each known algorithm can be constructed from some combinations of components found at the lowest hierarchy of the categorization tree. Also, new algorithms to enforce serializability could conceivably arise from appropriately selected combinations of components. Furthermore, the categorization scheme not only serves as a tool for the exploration of new algorithms, but it also can be used as a template for constructing or fine-tuning an algorithm to fit the specific database system under consideration.

The synthesis of concurrency control algorithms, although deemed a minor purpose of this research, nevertheless presents some insights to the possibility of systematic synthesis of algorithms based on the mathematical properties of classes of execution sequences and on the
algorithms of more containing classes.

C. Further Research

There are two major areas for further research.

1. Although the categorization scheme offers a way to compare algorithms qualitatively, the quantification of various components in the categorization tree has to be established as the next step of refinement. The quantification will not be easy considering the infancy of this field.

2. The underlying framework for the categorization is extensively simplified due to the extreme complexity of distributed database systems. Immediate relaxations on the framework can be listed as follows;

- **Reliability issues**  
  Examples concern the node and communication failures, and the problem of network partitioning. At the current state of the art, the theoretical groundwork is still limited.

- **Data semantics**  
  The underlying framework assumes that data are independent of each other. The relaxation will allow consistency constraints as superstructures on the data.
• **Transaction semantics**  As observed by Papadimitriou (1979), special types of transactions, such as copiers, can simplify the requirements on the concurrency control algorithms.

• **Heterogenous locking**  Either the hierarchical locking technique or the level of isolation, which is expressed by variations in locking (shared, exclusive), can have significant impacts on the categorization scheme.

The above relaxations, together with the study of new algorithms and extension of theoretical works, could possibly change or expand the proposed categorization.
VIII. BIBLIOGRAPHY


Corporation of America, Cambridge, Massachusetts, January 1980a.


Kaneko, A., Y. Nishihara, K. Tsuruoka and M. Hattori. "Logical Clock Synchronization Method for Duplicated Database Control." IEEE First International


Miranda, S. M. "Specification and Validation of Two Ring-


