TITLE OF REPORT
Non-Serializability with Wander-Transactions in Skeleton-Databases

DATE
1988-12-30

NO. OF PAGES AND APP.
444

AUTHOR(S)
Mads Nygård

GROUP
Database Technology

PROJECT NO.
148109.04

EXTRACT
This Dr.Techn.-thesis introduces a new division between global and local correctness criteria for distributed databases. It contains a thorough analysis and discussion of the possibilities and effects of different correctness criteria globally and locally. It also presents an interesting application with some non-traditional database and transaction types which suit the division and exploit the analysis and discussion. The new type of distributed database, the skeleton-database, is characterized by the lack of specific integrity constraints. The new type of distributed transaction, the wander-transaction, is characterized by the existence of specific overall semantics information. A new correctness criterion ASER*, or extended alternative-to-serializability, emerges corresponding to a combination of these two concepts. This breaks both with the usual serializability criterion and the usual recoverability criterion. The main emphasis is on the non-serializability aspect. The essence is that the parallelism increases - without reduced safety.

3 INDEXING TERMS
Distributed Computing Distribuert databeh.
Database Systems Database-systemer
Serializability Theory Serialiserings-teori

Steinar H. Kvitsand
Director
### CONTENTS

Preface \hspace{1cm} vii  
Acknowledgments \hspace{1cm} ix  
Summary \hspace{1cm} xi  

Part I \hspace{1cm} 1  
1 Introduction \hspace{1cm} 3  
   1.1 Overview \hspace{1cm} 4  
   1.2 Assumptions \hspace{1cm} 6  

Part II \hspace{1cm} 11  
2 Centralized Database Systems \hspace{1cm} 13  
   2.1 Theme \hspace{1cm} 14  
   2.2 Theory \hspace{1cm} 17  
   2.3 Concurrency Control \hspace{1cm} 65  
   2.4 Recovery \hspace{1cm} 81  
3 Distributed Database Systems \hspace{1cm} 89  
   3.1 Theme and Theory \hspace{1cm} 90  
   3.2 Fragmented but Non-Replicated Systems \hspace{1cm} 112  
   3.3 Fragmented and Replicated Systems \hspace{1cm} 121  
   3.4 System Examples \hspace{1cm} 127  

Part III \hspace{1cm} 129  
4 Classes of Serializability \hspace{1cm} 131  
   4.1 Transaction Model \hspace{1cm} 132  
   4.2 Classes of Schedules \hspace{1cm} 135  
   4.3 Class Relations and Facts \hspace{1cm} 160  
5 Levels of Non-Serializability \hspace{1cm} 179  
   5.1 Transaction Model \hspace{1cm} 180  
   5.2 Consequence Analysis \hspace{1cm} 183  
   5.3 Consistency Hierarchy \hspace{1cm} 185
PREFACE

This thesis is submitted to the Dr.Techn.-degree at the Norwegian Institute of Technology, the University of Trondheim.

The work has been mainly financed by the Computing Centre at the University of Trondheim, the Foundation for Scientific and Industrial Research at the Norwegian Institute of Technology.
ACKNOWLEDGMENTS

I need to thank some people and institutions for their support in my work with this thesis.

First, I am deeply grateful to the two employers that have funded me throughout the last few years, the Norwegian Institute of Technology and the Computing Centre at the University of Trondheim.

Next, the value of the everlasting backing of my two superiors Managing Director Karl G. Schjetne and Managing Director Steinar H. Kvitsand, the Computing Centre at the University of Trondheim, cannot be overestimated.

Last, I want to reward my parents Astrid and Jens for making it all possible.
SUMMARY

The basic highlights of this thesis are the following:

I) A **new division** between global and local correctness criteria for distributed databases is introduced. Globally means per database, while locally means per site or per item. This opens the way for specific systems to have more appropriate correctness criteria.

II) A **thorough analysis and discussion** of the possibilities and effects of different correctness criteria globally and locally is carried out.

III) An **interesting application** with some non-traditional database and transaction types is presented. This application suits the division indicated in I) and exploits the analysis and discussion mentioned in II).

IV) An **in-depth investigation and discussion** of the consequences and characteristics of the correctness criteria specified for this application type is performed.

V) A **parallel treatment** of the correctness criteria for another application area is given. This supplements the basic case and increases the understanding of how different correctness criteria are related.

Some of the corresponding concepts and results are the following:

This thesis introduces the **skeleton-database** concept. This is a new type of distributed database. It contains parts which have the same schema without having the same content. However the local databases do not merely constitute the results from horizontal fragmentation operations on a global database. The items of corresponding fragments in such a traditional distributed database only have the same logical type. While the items of corresponding parts in our alternative skeleton-database both have the same logical type and represent variants of the same physical entity. A skeleton-database is characterized by the lack of specific integrity constraints. This opens the way for the definition of alternative and additional correctness criteria instead of and on top of the preservation of integrity constraints.

This thesis also introduces the **wander-transaction** concept. This is a totally new type of distributed transaction. It traverses a skeleton-database trying to seize an optimal set (complying with any combined conditions on the attribute-values) of specific database items. A wander-transaction is characterized by the existence of specific overall semantics information. This leads the way to the specification of concrete correctness criteria instead of and on top of the preservation of integrity constraints.
A new correctness criterion ASER, or alternative-to-serializability, emerges corresponding to skeleton-databases only. It breaks with the usual serializability criterion (one-after-the-other atomicity). As a result, true parallel behaviour may now still give sensible results.

The new correctness criteria ASER*, or extended alternative-to-serializability, emerges corresponding to a combination of skeleton-databases and wander-transactions. This breaks both with the usual serializability criterion (one-after-the-other atomicity) and the usual recoverability criterion (all-or-nothing atomicity). (It even breaks with the usual atomic commitment criterion; i.e. all-or-none atomicity/unity). As a result, both true parallel behaviour and true partial behaviour may now still give sensible results. The primary effect is the true parallel behaviour. Hence the main emphasis is on the non-serializability aspect.

All write-read conflicts among transactions are recorded in binary relations typically named WR. Further all read-write conflicts among transactions are recorded in binary relations typically named RW. And all write-write conflicts among transactions are recorded in binary relations typically named WW. The usual correctness criterion SER, or serializability, corresponds to the WR-, RW- and WW-relations all being consistent partial orders globally (i.e. database wise). This also induces the same requirement locally (i.e. item or site wise).

A read-before-write restriction says that every item which is to be written by a transaction, first has to be read by it. This case corresponds directly and naturally to the combination of skeleton-databases and wander-transactions. Here ASER* requires that the WR-relation is to be a partial order globally. It only requires that the RW- and WW-relations are to be consistent partial orders locally - though also consistent with each local WR-order.

The not-read-before-write freedom says that no item which is to be written by a transaction, first has to be read by it. This case opens the way for a natural and direct generalization/extension of our alternative specifications. Here "ASER*" requires that the WR- and RW-relations are to be consistent partial orders globally. It only requires that the WW-relation is to be a partial order locally - though also consistent with each local WR- and RW-order.

The essence of both these cases is that the parallelism increases - without reduced safety. Non-serializability is allowed - whilst still achieving correctness.
Part I
1 Introduction

This first chapter gives a general survey of the thesis. The different chapters are examined with respect to their main content, scope and necessity. We also indicate the minimal background needed by a reader.
1.1 Overview

The thesis consists of 10 chapters, divided into 4 parts, plus an appendix.

Part I, Chapter 1, contains this introduction only.

Part II, Chapters 2 and 3, is a state-of-the-art presentation of a specific part of the database systems field. The emphasis is on transaction processing with special focus on parallel use and erroneous behaviour. Chapter 2 covers the resulting issue of atomic transactions in centralized environments. Chapter 3 covers the same issue in distributed environments, including a general treatment of distributed systems per se.

The aim of Part II is to set the scene for the next two parts. It establishes a framework to work within. It also creates a basis to work from.

Part III, Chapters 4 and 5, gives a thorough theoretical discussion of two subfields of transaction processing, serializability and non-serializability. The purpose is to relate known facts and existing ideas from these two orthogonal directions. Chapter 4 mainly deals with different ways of achieving serializable transactions. While Chapter 5 mainly deals with different consequences of allowing non-serializable transactions.

The goal of Part III is to analyze two separate dimensions to be treated together in the next part. It pinpoints the efficiency effects and correctness effects of different methods of processing transactions.

Part IV, Chapters 6, 7, 8, 9 and 10, contains an in-depth examination of our new contribution to the field. Chapters 6 and 7 introduce two new concepts, the skeleton-database and the wander-transaction. One of the characteristics of skeleton-databases is the lack of specific integrity constraints. While one of the characteristics of wander-transactions is the existence of specific overall semantics information. Chapters 8 and 9 analyze the possibilities inherent in the specifications of these new concepts, both overall and in detail. One of the major results of combining the two concepts is the emergence of different types of non-serializable transactions which still do not corrupt the database or transaction correctness. Chapter 10 discusses further refinements of the notions of skeleton-databases and wander-transactions, their corresponding theory and its application.

The essence of Part IV is to show how a specific solution to a specific problem, instead of the general solution to the general problem, gives increased efficiency without reduced correctness. Non-serializability is allowed with wander-transactions in skeleton-databases. Hence fewer reads and writes have to wait; i.e. more parallelism is achieved. The system, a database and its transactions, still behaves correctly.
Actually wander-transactions in skeleton-databases is not the single main point of our contribution. First, Part IV introduces a new division between global and local correctness criteria for distributed databases - where globally means per database, and locally means per site or per item. It contains a thorough analysis and discussion of the possibilities and effects of different correctness criteria globally and locally. Next, Part IV presents wander-transactions in skeleton-databases as an interesting application with some non-traditional database and transaction types which suit the division and exploit the analysis and discussion. It contains an in-depth investigation and discussion of the consequences and characteristics of the correctness criteria specified for this application type. Last, Part IV gives a parallel treatment of the correctness criteria for another application area which supplements the basic case. Hence wander-transactions in skeleton-databases is more a general guideline for a theoretical exploration of correctness criteria for distributed databases.

The Appendix presents some extra material needed in Chapters 4 and 9.

Note that this thesis is more biased towards principles and theoretical aspects than performance and practical issues. An exception is where practical concerns direct theoretical decisions.
1.2 ASSUMPTIONS

A reader of this thesis is anticipated to have a knowledge of centralized database systems at the level of [Kort86] or [Date86], of computer networks at the level of [Sta85] or [Tane81], and of operating systems at the level of [Pete85] or [Habe76].

It is also anticipated that the reader has a basic knowledge of mathematical logic (see for example [Hami88]), set theory (see for example [Stan77]) and graph theory (see for example [Bond76]). A unified but non-complete presentation of these three fields is given in [Gill76].

As the terminology used in these three fields varies a lot, the following is a presentation of the definitions and notations that we will refer to and employ.

First, we will look at some aspects of mathematical logic.

A universal proposition is stated as:
- \( \forall x (\in S_x), y (\in S_y), \ldots z (\in S_z) \) [logical-expression]
The logical-expression is to be true for each and all \( x, y, \ldots \) and z.

An existential proposition is stated as:
- \( \exists x (\in S_x), y (\in S_y), \ldots z (\in S_z) \) [logical-expression]
The logical-expression is to be true for at least one \( x, y, \ldots \) and z.

The parentheses on the set-membership statements indicate that such statements will be excluded where appropriate.

Only \( \bot \) is used for negation, both \( \Delta \) and "and" are used for conjunction, and both \( \lor \) and "or" are used for disjunction. Further only \( \Rightarrow \) is used for implication, but both \( \equiv \) and "iff" are used for equivalence.

Next, we will look at some aspects of set theory.

A set is a collection of entities:
- \{a, b, c, \ldots\}

A permutation of a set is any sequence of all its members like:
- a, b, c, \ldots
A **binary relation** is a set of 2-tuples of entities:

- $\{(a, b) \mid \text{logical-expression}\}$

Such a set $S$ of ordered pairs has a shorthand notation:

- $a \preceq b \iff (a, b) \in S$

A **partial order** is a binary relation $B$ so that:

- $\forall a \left( (a, a) \notin B \right)$ (a)
- $\forall a, b \left( (a, b) \in B \implies (b, a) \notin B \right)$ (b)

and

- $\forall a, b, c \left( [(a, b) \in B \land (b, c) \in B] \implies (a, c) \in B \right)$ (c)

a) is the irreflexive property. b) is the asymmetric property. c) is the transitive property.

A **total order** is a partial order $P$ where all the entities are ordered:

- $\forall a, b \left( (a, b) \in P \lor (b, a) \in P \right)$

A **partially ordered set** is an ordered pair $(S, P)$ of a set $S$ and a partial order $P$ so that:

- $\forall (a, b) \in P \left[ a \in S \land b \in S \right]$

A **totally ordered set** is a partially ordered set $(S, P_S)$ where the partial order $P_S$ is a total order.

A partially ordered set $(S', P_{S'})$ is a **rejection on** $S'$ of a partially ordered set $(S, P_S)$ iff:

- $S' \subseteq S$ (a)

and

- $\forall a \in S', b \in S' \left[ (a, b) \in P_S \implies (a, b) \in P_{S'} \right]$ (b)

A partially ordered set $(S', P_{S'})$ is a **prefix** of a partially ordered set $(S, P_S)$ iff:

- $(S', P_{S'})$ is a rejection-on-$S'$-of $(S, P_S)$ (a)

and

- $\forall b \in S' \left[ \forall a \in S \left[ (a, b) \in P_S \implies a \in S' \right] \right]$ (b)
Last, we will look at some aspects of graph theory.

A **directed graph** is an ordered pair \((V, A)\) of a set \(V\) and a binary relation \(A\) so that:

- \(\forall (a, b) \in A \ [a \in V \land b \in V]\)

The \(V\)-set contains the vertices, and the \(A\)-relation contains the arcs. The shorthand notation for arcs is:

\[a \rightarrow b \iff (a, b) \in A\]

A **directed graph-set** is a "directed graph" \((V, A)\) where the "binary relation" \(A\) may contain alternative arc-pairs. An alternative arc-pair

\[a \rightarrow b \lor c \rightarrow a\]

means that either \(a \rightarrow b\) or \(c \rightarrow a\) but not both applies.

A directed graph-set corresponds to a set of directed graphs, one for each possible combination of choices from the alternative arc-pairs.

A **directed acyclic graph** is a directed graph \((V, A)\) where the binary relation \(A\) is a partial order.

A directed acyclic graph \((V, A)\) thus mirrors a partially ordered set \((S, P_S)\) where:

\[S = V \land P_S = A\]

A **non-directed graph** is an ordered pair \((V, E)\) of a set \(V\) and a binary relation \(E\) so that:

- \(\forall (a, b) \in E \ [(b, a) \in E \land a \in V \land b \in V]\)

The \(V\)-set contains the vertices, and the \(E\)-relation contains the edges. The shorthand notation for edges is:

\[a \leftrightarrow (a, b) \in E \land (b, a) \in E\]

A non-directed graph \((V, E)\) thus mirrors a directed graph \((V, A)\) where:

\[\forall a \rightarrow \exists b \ [a \rightarrow A b \land b \rightarrow A a]\]

A **path** in a graph \((V, A/E)\) is a sequence of arcs/edges of the type:

- \((a, b), (b, c), (c, d), \ldots (e, f)\)

A **cycle** in a graph \((V, A/E)\) is a sequence of arcs/edges of the type:

- \((a, b), (b, c), (c, d), \ldots (e, a)\)
A cycle is thus a path starting and ending at the same vertex. A directed acyclic graph is further a directed graph with no cycles, while a directed cyclic graph is a directed graph with at least one cycle.

A topological sort of a directed acyclic graph \((V, A)\) is a sequence of all its vertices so that for any

\[
... a, ... b, ...
\]

occurring in the sequence, there is no path from \(b\) to \(a\) in the graph.

Note that a lot of graphs will be shown in this thesis. Basically they are used as a proof mechanism. In parallel they play an even more important role as an illustration tool.
PART II
2 Centralized Database Systems

In this chapter we present our interpretation of the notion of transaction processing in centralized databases. The main issues are consistency preservation and reliability assurance in the presence of interference and failures. Hence concurrency control and recovery are the main subjects to be treated. An overall treatment of transaction processing in such systems may be found in [Date83] and [Ullm82], while [Bern87b] and [Papa86] cover concurrency control and recovery in these systems in detail.
2.1 **Theme**

This section contains an overall introduction to the basic concepts and notions of this chapter.

2.1.1 **Transaction Processing Issues**

The main issue to be dealt with is transaction processing on common data resources. We will discuss both centralized environments and distributed environments. We will focus on both parallel use and erroneous behaviour.

The distinction between centralized and distributed systems corresponds to the existence or lack of so-called signal observability, see Section 3.1.1. Parallel use refers to interleaved operations on data resources, while erroneous behaviour concerns software or hardware faults of operations and data resources. One of the earliest contributions to this field came with [Bjor72]. One of the most influential ones came with [Gray78].

In operating systems the basic notion is atomic actions. It corresponds to implementing a single read and/or write on a single data-item as the logically isolated unit. In database systems the basic notion is atomic transactions. This is a generalization of atomic actions. It corresponds to implementing multiple reads and/or writes on multiple data-items as the logically isolated unit. This concept stems from [Lome77] - though under another name. Discussions of the concept per se and its implementation may be found in [Gray81a], [Trai82] and [Spec83]. These relate both to centralized and distributed databases.
2.1.2 Database Systems Model vs. Operating Systems Model

It is the differences between database systems and operating systems that lead to the need of atomic transactions instead of only atomic actions. A comparison of the two corresponding models is given in Table 2.1.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Operating Systems Model</th>
<th>Database Systems Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processes</td>
<td>Related (Task Specification)</td>
<td>Non-related</td>
</tr>
<tr>
<td>Resources</td>
<td>Non-related</td>
<td>Related (Integrity Constraints)</td>
</tr>
<tr>
<td>Amount</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Stability</td>
<td>Fixed</td>
<td>Non-fixed</td>
</tr>
<tr>
<td>Addressing</td>
<td>By Name</td>
<td>By Content</td>
</tr>
<tr>
<td>Decidability</td>
<td>Static</td>
<td>Dynamic</td>
</tr>
</tbody>
</table>

Processes indicate: Computational units = Executions of programs  
Resources indicate: Accessible units = Data-set  
Amount indicates: Number of existing resources  
Stability indicates: Is the set of resources stable or not?  
Addressing indicates: Way of indicating resources  
Decidability indicates: When is the mapping from logical indication to physical resources decidable?

It is the resources that are related in the database systems model. They are coupled through integrity constraints, see Section 2.2.1. In the operating systems model it is the processes that are related. The coupling depends on the semantics of the task to be implemented. The database systems model also works with a larger number of resources than the operating systems model. The number of resources is even rapidly varying over time. Further the designation of the resources to be accessed is mainly based on resource-contents in the database systems model. The designation of resources is even only interpretable at run-time. In the operating systems model the designation of the resources to be accessed is mainly based on resource-names. The two resulting systems models are illustrated in Figs. 2.1 and 2.2 respectively.

The extent to which a database system may use low-level services made available by the operating system is discussed in general in [Ston81]. The applicability of the low-level services of Unix specifically is discussed in [Wein82].
Fig. 2.1. Processes and resources in the Operating Systems Model.

Fig. 2.2. Processes and resources in the Database Systems Model.
2.2 Theory

This section contains an in depth elaboration of the basic concepts and notions of this chapter.

2.2.1 System Database and Transaction Programs Definitions

We will use the following informal descriptions:

A system database comprises a set of items \((x_1,x_2,...,x_m)\) which are the accessible units of the database. Their values \((v_1,v_2,...,v_m)\) which constitute the current state of the database, are supposed to satisfy certain integrity constraints. These are different rules restricting the sets of possible values that may coexist.

As an example,

\[ v_m = \sum_{i=1}^{m-1} v_i \]

\((m\) being the number of items) says that the last item is always assumed to contain the sum of all the other item values.

The transaction programs represent a set of computations. Each transforms a system database from one state to another. First it reads some specific items. Next it computes some new item values from the old item values. Last it writes the corresponding items.

A transaction is a single execution of a transaction program.
2.2.2 TRANSACTION PROCESSING MODEL

A simple model for the processing of transactions is shown in Fig. 2.3.

![Diagram of Transaction Processing Model]

Fig.2.3. Model of Transaction Processing.

A typical transaction $T_i$ corresponds to a transaction program having five parts. $G_i$ - for get message - enters some values from the keyboard of a terminal. $R_i$ - for read values - retrieves the contents of some items in the database. $C_i$ - for compute data - calculates the new values to be written or printed from the old values read or entered. $W_i$ - for write values - updates the contents of some items in the database. $P_i$ - for put message - prints some values on the screen of a terminal.

A terminal user that wants to access the database, will initiate a transaction and give some parameters to it through a message entered by a get. The statically given transaction program usually indicates how the new values are to be calculated from the old values in the compute part. While the dynamically given transaction parameters usually indicate which items are to be retrieved in the read part - and which items are to be updated in the write part. The transaction will terminate itself and give some results to the terminal user through a message printed by a put.

(We have only included one read-action and one write-action per transaction in the above figure due to the limited amount of space. Likewise we have not included any indication of which items are retrieved and/or updated. Thus the illustrated model corresponds more to a special model to be employed in Chapter 4, than to the general model to be employed in this chapter. The general model actually
allows several read-actions and/or several write-actions per transaction).

The transaction processing \textit{problems} to be dealt with in this context are two-fold:

- \textit{Interference} from parallel use of the database leads to concurrency control problems.

- \textit{Failures} from erroneous behaviour of the database - software or hardware - leads to recovery problems.

Thus the transaction processing goal of achieving atomic transactions will also be treated as two-fold. The two following \textit{subgoals} are usually considered as the correctness criteria:

- One-after-the-other Atomicity:

  The effect of handling several transactions being run in parallel should be as if they were executed in some - unknown - serial order. This is carried out by a concurrency control mechanism.

- All-or-nothing Atomicity:

  The effect of dealing with one transaction being stopped in the middle due to a software or hardware error should be as if it was not started at all - or as if it was totally finished. This is carried out by a recovery mechanism.

The employed nomenclature is our own. The terms commonly used in the literature for the two subgoals are \textit{serializability} and \textit{recoverability}. The two subcases are illustrated in Figs. 2.4 and 2.5 respectively.

![Diagram](image-url)

**Fig. 2.4.** The \textbf{One-After-The-Other} effect of transactions execution.
(In the two above figures we have concentrated on the read- and write-actions of a transaction as given in Fig. 2.3. But in contrast with Fig. 2.3 we have now included more than one read-action and more than one write-action per transaction. This is in correspondence with the general model to be employed in this chapter. However we have still not included any indication of which items are retrieved and/or updated due to the limited amount of space).

Before we continue our treatment of concurrency control and recovery issues, it is appropriate to mention some units related to the processing of transactions. According to [Lync83] we may differentiate between the following three units:

- **Unit of Work**

  How many old item values should a specific transaction be able to use as a basis for the computation of the new item values?

  This concerns how many previously read items may be used in the calculation of values to be written.

- **Unit of Concurrency**

  How many new item values should be made available as a whole from one specific transaction to all other transactions?

  This concerns how many currently written values must be made collectively visible.

- **Unit of Recovery**

  How many new item values of a specific transaction should eventually either be undone or redone?

  This concerns how many values being written during a breakdown should be arranged as if none of them did happen - or arranged as if all of them did happen.
These three units are shown in Figs. 2.6, 2.7 and 2.8 respectively.

![Diagram of Work Unit](image1)

**Work Unit**

Fig. 2.6. Illustration of the Work Unit.

![Diagram of Concurrency Unit](image2)

**Concurrency Unit**

Fig. 2.7. Illustration of the Concurrency Unit.

![Diagram of Recovery Unit](image3)

**Recovery Unit**

Fig. 2.8. Illustration of the Recovery Unit.

The three illustrated units are often but not necessarily the same unit - termed transaction.

(In the three above figures we have again concentrated on the read- and write-actions of a transaction as given in Fig. 2.3. But in contrast with Figs. 2.4 and 2.5 we have now included some indication of which items are retrieved and/or updated).
2.2.3 Centralized Database Systems Model

A simple model of the parts of a centralized database system is shown in Fig. 2.9.

The transaction processes represent computational machines. Each carries out a single execution of a transaction program. The three manager processes implement different system functions. The transaction manager is an overall supervisor controlling several transaction processes. The concurrency manager and recovery manager respectively implement the one-after-the-other atomicity and the all-or-nothing atomicity.

First, let us concentrate on the concurrency manager. It is detailed in Fig. 2.10.

The system scheduler acts as an orderer and timer of database operations. The two types of schedules contain different sets of reads and writes. The transaction schedules represent a set of mutually unordered sequences of reads and writes, while the system schedule represents a single sequence of reads and writes. A further treatment of the pragmatics of concurrency managers will be given in Section 2.3.

Second, let us concentrate on the recovery manager and its accompanying buffer manager. They are detailed in Fig. 2.11.

The recovery manager records the effects and carries out the intentions of database operations, while the buffer manager implements virtual fast access to real slow database and log storage. The system cache constitutes a fast buffer of database and log items. The system log contains a sequential description of database operations. A
Fig. 2.10. Function of the Concurrency Manager.

Fig. 2.11. Function of the Recovery Manager.

Further treatment of the pragmatics of recovery and buffer managers will be given in Section 2.4.
2.2.4 TRANSACTION SCHEDULE AND SYSTEM SCHEDULE

Let us now formalize the notions of a transaction schedule and system schedule from the previous section. The notations used here have been inspired by [Bern87b].

As in Section 2.2.1, a typical item will be denoted \( x \). As in Section 2.2.2, a retrieval or update of such an item will be given by respectively \( R_i(x) \) - for read of item \( x \) by transaction \( i \), and \( W_i(x) \) - for write of item \( x \) by transaction \( i \). Successful termination of a transaction will be indicated by \( C_i \) - for commit of transaction \( i \), while unsuccessful termination of a transaction will be indicated by \( A_i \) - for abort of transaction \( i \). The significance of the last two concepts will become clear later. (Do not confuse commit with compute from Section 2.2.2 - though both have the same abbreviation).

**System database** is a set \( D \): 

\[ D = \{ x | x \text{ item} \} \]  

(Eq. 2.1)

**Transaction schedule** \( T_i \) is a Partially Ordered Set \( (t_i, <_i) \): 

\[ t_i \subseteq \{ R_i(x) | x \in D \} \cup \{ W_i(x) | x \in D \} \cup \{ C_i, A_i \} \]  

(Eq. 2.2)

\[ C_i \in t_i \iff A_i \notin t_i \]  

(Eq. 2.3)

\[ [ e \in t_i \land e \in \{ C_i, A_i \} \land p \in t_i \land p \neq e ] \implies p <_i e \]  

(Eq. 2.4)

\[ [ R_i(x) \in t_i \land W_i(x) \in t_i ] \implies [ R_i(x) <_i W_i(x) \lor W_i(x) <_i R_i(x) ] \]  

(Eq. 2.5)

Eq. 2.1 defines the accessible units, while Eqs. 2.2 to 2.5 describe a computational unit. The operational elements are defined in Eq. 2.2 - with the restriction given in Eq. 2.3 that either a commit or abort is included. Eq. 2.4 states that the commit or abort must succeed all other operational elements, and Eq. 2.5 states that a read and write on a common accessible unit must be ordered.

To illustrate the partially ordered set \( T_i \), we may draw it as a directed acyclic graph (DAG) with vertices corresponding to \( t_i \) and arcs corresponding to \( <_i \). Some examples are given in Fig. 2.12. If the partial order of \( T_i \) even is a total order, we may drop the arcs from the DAG. This is shown in Fig. 2.12 for \( T_1, T_2 \) and \( T_3 \).

It is sensible to allow maximum one read and/or one write per item per transaction, and not to allow a read to succeed a write in order on a common item in a transaction. (This is a further restriction of Eq. 2.5). If not, we would have redundant operational elements. To allow a
write to succeed a read in order on a common item in a transaction, is both sensible and important. If a write on a specific item always must be preceded by a read on the same item, we have the so-called **read-before-write case**. If this is not a restriction — i.e., a write on a specific item must not always be preceded by a read on the same item, we have the so-called **not-read-before-write case**. A single read without a succeeding write is called a **read-only**, while a single write without a preceding read is called a **blind-write**.

**Conflicting operations** is a Binary Relation $\sim$:

\[
\forall x \in D \quad [i \neq j \Rightarrow \left( R_i(x) \sim W_j(x) \land W_i(x) \sim R_j(x) \land W_i(x) \sim W_j(x) \right)]
\]  

(Eq. 2.6)

**Complete system schedule** $CH$ is a Partially Ordered Set $(h, \prec)$:

- $h = \bigcup_{i=1}^{n} t_i$  

(Eq. 2.7)

- $\prec \supseteq \bigcup_{i=1}^{n} \prec_i$  

(Eq. 2.8)

- $\forall p \in h, q \in h \left[ p \sim q \Rightarrow [p \prec q \lor q \prec p] \right]$  

(Eq. 2.9)
System schedule $H$ is any prefix of a complete system schedule $CH$.

Committed projection $C(H)$ of a system schedule $H$ is the restriction of $H$ on $U_{C_1}H_{t_1}$.

Eq. 2.6 states that one read and one write or two writes from separate computational units on a common accessible unit are \textit{ <--}-related, while Eqs. 2.7 to 2.9 describe the system unit. The operational elements are defined in Eq. 2.7, and the ordering pairs are defined in Eq. 2.8. Eq. 2.9 states that \textit{ <--}-related operational elements must be ordered.

Again to illustrate the partially ordered sets $CH, H$ and $C(H)$, we may draw each as a \textit{ directed acyclic graph (DAG) } with vertices corresponding to $h$ and arcs corresponding to <. Some examples are given in Fig. 2.13. $H_1$ is a prefix of $CH_1$, and $C(H_1)$ is the committed projection of $H_1$. If the partial order of $CH, H$ or $C(H)$ even is a total order, we may drop the arcs from the DAG. This is shown in Fig. 2.13 for $CH_2$ and $C(H_1)$.

Fig. 2.13. Examples of System Schedules.

The reason for using $H$ as a symbol for a system schedule is partly practical and partly historical. The most natural symbol, $S$, will be
used to denote another concept. In Chapter 4 we will also analyze some older source-materials concerning this field. Those are [Bern79b] which employed the notion log, and [Papa79] which employed the notion history. The newer reference-materials [Bern87b] and [Papa86] use respectively history and schedule.
2.2.5 Concurrency Control and Recovery Problems

The basic problems relating to concurrency control and recovery may be divided into four classes:

i) Lost Update

ii) Inconsistent Retrievals

iii) Update Dependent on Uncommitted Update

(eventually leading to a lost update at an abort)

iv) Retrieval Dependent on Uncommitted Update

(eventually leading to an invalid retrieval at an abort)

Cases i) and ii) represent concurrency control problems, while cases iii) and iv) represent recovery problems - that eventually may lead to concurrency control problems.

First, a typical lost update situation is illustrated in Fig. 2.14.

![Diagram](image)

Fig. 2.14. Update \( W_1(x) \) gets lost at update \( W_3(x) \).

The bracketed operation \( R_1(x) \) is included only to clarify the effects. It may be deleted without changing the basic situation.

As an example, the initial value of the referenced item is supposed to be

\[ x_5 = 2, \]

and the semantics of the two existing transactions are assumed to be:

\[ T_1 = x := x + 3 \]

\[ T_2 = x := x + 4 \]

The two possible serial executions corresponding to one-after-the-other atomicity would have led to the following final value for the referenced item:

\[ T_1 \circ T_2 \Rightarrow x_f = 2 + 3 + 4 = 9 \]

\[ T_2 \circ T_1 \Rightarrow x_f = 2 + 4 + 3 = 9 \]
As it is, first \( W_1(x) \) updates \( x \) to 2+3=5, then \( W_2(x) \) updates \( x \) to 2+4=6. Thus the final value becomes

\[ x_f = 6 \]

being different from the result in both the quoted serial executions. Thus the given execution must be considered incorrect - stemming from the loss of update \( W_1(x) \). Its effect has vanished from the database state.

Second, a typical inconsistent retrievals situation is illustrated in Fig. 2.15.

As an example, the initial values of the referenced items are supposed to be

\[ x_s = 2 \]
\[ y_s = 3, \]

and the semantics of the two existing transactions are assumed to be:

\( T_1 = \text{print } x; \text{ print } y \)

\( T_2 = x := 4; \text{ } y := 5 \)

The two possible serial executions corresponding to one-after-the-other atomicity would have led to the following printed and final values for the referenced items:

\( T_1 \circ T_2 \rightarrow x_p = 2; \text{ } y_p = 3; \text{ } x_f = 4; \text{ } y_f = 5 \)

\( T_2 \circ T_1 \rightarrow x_p = 4; \text{ } y_p = 5; \text{ } x_f = 4; \text{ } y_f = 5 \)

As it is, first \( R_1(x) \) retrieves 2 for \( x \), next \( W_2(x) \) and \( W_2(y) \) update \( x \) and \( y \) to 4 and 5, then \( R_1(y) \) retrieves 5 for \( y \). Thus the printed and final values become

\[ x_p = 2; \text{ } y_p = 5; \text{ } x_f = 4; \text{ } y_f = 5 \]

with the printed values being different from the results in both the quoted serial executions. Even though the final values are equal to the results in both the quoted serial executions, the given execution
must be considered incorrect - stemming from the inconsistency between retrievals \( R_1(x) \) and \( R_1(y) \). They belong to two different database states.

Third, a typical update dependent on uncommitted update situation is illustrated in Fig. 2.16.

![Fig. 2.16. Update \( W_1(x) \) gets dependent on uncommitted update \( W_2(x) \) and gets (possibly) lost at abort \( A_2 \).](image)

As an example, the initial value of the referenced item is supposed to be

\[ x_s = 2, \]

and the semantics of the two existing transactions are assumed to be:

\[ T_1 = x := 3 \]
\[ T_2 = x := 4 \]

The two possible total (and serial) executions corresponding to all-or-nothing atomicity would have led to the following final value for the referenced item:

\[ T_2 \circ T_1 \rightarrow x_f = 4, 3 \]
\[ T_1 \rightarrow x_f = 3 \]

As it is, first \( W_2(x) \) updates \( x \) from 2 (the before-image) to 4 (the after-result), next \( W_1(x) \) updates \( x \) from 4 (the before-image) to 3 (the after-result), then \( A_2 \) aborts \( T_2 \). A simple abortion-strategy for \( T_2 \) will lead to an exchange of its after-result 4 for \( x \) with its before-image 2 for \( x \). This way the final value becomes

\[ x_f = 2 \]

being different from the result in both the quoted total executions. In this case the given execution must be considered incorrect - stemming from the loss of update \( W_1(x) \) (having been dependent on an uncommitted update \( W_2(x) \) whose transaction later aborted). Its effect has vanished from the database state.
Fourth, a typical retrieval dependent on uncommitted update situation is illustrated in Fig. 2.17.

![Diagram](image)

Fig. 2.17. Retrieval $R_1(x)$ gets dependent on uncommitted update $W_2(x)$ and gets invalid at abort $A_2$.

As an example, the initial value of the referenced item is supposed to be

$$x_0 = 2,$$

and the semantics of the two existing transactions are assumed to be:

$$T_1 = \text{print } x$$

$$T_2 = x := 3$$

The two possible total (and serial) executions corresponding to all-or-nothing atomicity would have led to the following printed and final values for the referenced item:

$$T_2 \circ T_1 \rightarrow x_p = 3; x_f = 3$$

$$T_1 \rightarrow x_p = 2; x_f = 2$$

As it is, first $W_2(x)$ updates $x$ from 2 (the before-image) to 3 (the after-result), next $R_1(x)$ retrieves 3 for $x$, then $A_2$ aborts $T_2$. Any simple or advanced abortion-strategy for $T_2$ will lead to an exchange of its after-result 3 for $x$ with its before-image 2 for $x$. Any way the printed and final values become

$$x_p = 3; x_f = 2$$

with they being different from the results in both the quoted total executions. In any case the given execution must be considered incorrect - stemming from the invalidity of retrieval $R_1(x)$ (having been dependent on an uncommitted update $W_2(x)$ whose transaction later aborted). It belongs to a non-existing database state.
2.2.6 Concurrency Control Solutions

Here we will return to the concurrency control problems discussed in the previous section. The goal is to define and discuss different types of system schedules corresponding to different ways of preserving consistency. The approach employed is again inspired by [Bern87b].

Initially, let us illustrate the solutions to the concurrency control problems with informal descriptions.

A serial schedule is a complete system schedule where the transactions are executed strictly one by one (with no parallelism at all).

Two equivalent schedules are two system schedules where, for any starting values of each transaction \((G_1, G_2, ..., G_n)\) - i.e. the input messages, plus for any initial values of each item in the database \((v_1, v_2, ..., v_m)\), and for any functional computations of each transaction \((C_1, C_2, ..., C_n)\), we get the same final values of each item in the database \((v_1, v_2, ..., v_m)\), plus the same resulting values of each transaction \((P_1, P_2, ..., P_n)\) - i.e. the output messages. (See Fig. 2.3 in Section 2.2.2).

A serializable schedule is a system schedule with a specific set of transactions which is equivalent to one or another serial schedule of the same set of transactions. The effects are as if the transactions were executed strictly one by one in some or another way.

Actually each transaction is assumed to preserve the consistency of the database. This means that it will transform the database from one state satisfying the integrity constraints, into another state satisfying the integrity constraints, if executed in isolation.

This implies that the state-mapping of a serial schedule will preserve consistency. This again implies that the state-mapping of even a serializable schedule will preserve consistency.

Thus consistency preservation in executing transactions on a database may be secured by assuring that all executed schedules are serializable schedules. In Sections 2.2.7 and 4.3 we shall see that the implication only works one-way. The appropriate criterion corresponding to consistency preservation depends on the information available about the transactions and the database.

Observe the phrase "equivalent to one or another serial schedule" in the definition of a serializable schedule. If the outer world depends on a certain execution order of several transactions, this must be provided by a manual external handshake procedure. On the other hand, the inherent execution order of several actions within a single transaction will be provided by an automatic internal handshake procedure.

Most of the original work in this area came with [Eswa76]. In this chapter we will discuss two types of schedule equivalence, conflict equivalent schedules and (new) view equivalent schedules, respectively
leading to conflict serializable schedules and (new) view serializable schedules. Conflict equivalence and serializability correspond to the approach used in [Eswe76] and [Stea76], while the new view equivalence and serializability stem from [Yann84]. These two types of schedule equivalence are also covered by [Bern87b]. In Section 4.3 we will again discuss the two types together with two other types of schedule equivalence covered by [Papa86]. These are final-state equivalent schedules and (old) view equivalent schedules respectively leading to final-state serializable schedules and (old) view serializable schedules. ([Papa86] covers conflict equivalence and serializability too). We will even comment on (old) view equivalence and serializability where appropriate in this chapter. The old view equivalence and serializability were implicit in the work of [Stea76] and explicit in the work of [Rose82], while final-state equivalence and serializability stem from [Papa77]. The definitions of equivalent and serializable schedules quoted above effectively correspond to old view equivalence and serializability. The differences between these and the other equivalence and serializability variants will be clarified later. The terms old and new view serializable schedules have been introduced by us. [Hadz88] uses the nomenclature view serializability and commit view serializability.

Further, we will introduce some auxiliary concepts.

**Projection**

\[ H[\{T_i, T_j, \ldots, T_k\}] \]

of a system schedule \( H \) with regard to transactions \( T_i, T_j, \ldots \) and \( T_k \) is the Restriction of \( H \) on \( t_i \cup t_j \cup \ldots t_k \).

Thus commit projection (from Section 2.2.4) may be expressed:

\[-C(H) = H[\{T_i | C_i \in h\}] \quad (\text{Eq. 2.10})\]

**Concatenation**

\( T_i \circ T_j \circ \ldots T_k \)

of several transaction schedules \( T_i, T_j, \ldots \) and \( T_k \) is the Sequential Execution of first \( T_i \), then \( T_j, \ldots \) and last \( T_k \).

Thus equality to a sequential execution, serial schedule, may be expressed:

\( H = T_i \circ T_2 \circ T_3 \)

And equivalence to a sequential execution, serializable schedule, will be expressed:

\( H = T_i \circ T_2 \circ T_3 \)
Concatenation

\[ H_1 \circ H_2 \circ ... \circ H_k \]

of several system schedules \( H_1, H_2, \ldots \) and \( H_k \) is likewise the Sequential Execution of first \( H_1 \), then \( H_2 \), \ldots and last \( H_k \).

Next, we will give **formal definitions** of a set of system schedules allowing one kind of serializable executions with regard to interleaving.

First we need to formally specify the notion of a sequential execution.

**Serial schedules** is a Set \( S \):

\[
CH \in S \text{ iff }
- \ CH = T_i \circ T_j \circ ... \circ T_k
\]

(Eq. 2.11)

for \( i,j,\ldots,k \) being some or another Permutation of \( \{1,2,\ldots,n\} \)

Eq. 2.11 requires a total ordering of transactions (being much stronger than only a partial ordering of actions).

Then we may formally specify the equivalence and serializability notions.

**Conflict equivalent schedules** is a Binary Relation \( =_c \):

\[
H =_c H' \text{ iff }
- \ h = h'
\]

and

\[
\forall p \in t_i, q \in t_j
\]

(Eq. 2.12)

\[
[[p <_h q \land p \prec q \land A_i \notin h \land A_j \notin h]] \Rightarrow
p <_h q
\]

**Conflict serializable schedules** is a Set CSR:

\[
H \in \text{CSR iff }
- \ \exists \ CH \in S \ [C(H) =_c CH]
\]

(Eq. 2.14)
Eq. 2.12 says that the operational elements have to be the same, and Eq. 2.13 says that the action ordering of --related operational elements has to be preserved. Eq. 2.14 requires conflict equivalence of the committed projection of the schedule to some or another serial schedule.

It may be shown that this class of serializable schedules has a so-called **commit-projection-closed property**:

\[ \forall H \in CSR, \quad H' \quad \rightarrow \quad [H' \text{ is a prefix of } H \Rightarrow C(H') \in CSR] \]  

(Eq. 2.15)

(And not only \( C(H') \) will be a member in CSR, but also \( H' \) itself).

This is very important in **on-line scheduling**. At any time during execution any active transaction - i.e. one that is neither committed nor aborted, may fail and must thus be aborted. In worst case all the active transactions may eventually abort, leaving only the already committed transactions. Therefore all working criteria for delimiting schedules ought to correspond to the commit-projection-closed property.

To test class membership in CSR for a given system schedule \( H \), first find the committed projection \( H' = C(H) \), second make a directed graph \( CSG(H') \) as:

\[ V(CSG(H')) = \{T_i | t_i \in h'\} \]  

(Eq. 2.16)

\[ A(CSG(H')) = \]  

\[ \begin{align*}
\{T_j \rightarrow T_i| &\exists x \in D \ [W_j(x) < \_ R_i(x)] \} \quad \text{ (a)} \\
\cup \{T_j \rightarrow T_i| &\exists x \in D \ [R_j(x) < \_ W_i(x)] \} \quad \text{ (b)} \\
\cup \{T_j \rightarrow T_i| &\exists x \in D \ [W_j(x) < \_ W_i(x)] \} \quad \text{ (c)}
\end{align*} \]

Then \( H \in CSR \) if and only if \( CSG(H') \) is acyclic.

Cases a), b) and c) are all illustrated in Fig. 2.18. Observe that transitive arcs stemming from case c) between two and two transactions will be left out in the coming examples.

Note further that any topological sort of an acyclic \( CSG(H') \) corresponds to a serial schedule \( CH \) conflict equivalent to the committed projection \( C(H) \). Thus the given system schedule \( H \) may have several conflict serializations.

As examples for illustrating the conflict serializability concept, we will basically use the same schedules as in Section 2.2.5; i.e. \( H_1 \) and \( H_2 \) from Figs. 2.14 and 2.15 respectively.
First, let us start by respecifying $H_1$ as:

$$H_1 = \begin{align*}
T_1 & : R_1(x) & W_1(x) & C_1 \\
T_2 & : R_2(x) & W_2(x) & C_2
\end{align*}$$

(This is further an example of how we will represent a schedule in the text. Even when the schedule is a total order (see the comments concerning Fig. 2.13 in Section 2.2.4), we feel it is more illustrating to split the description among the involved transactions. Thus the horizontal axis represents "time", while the vertical axis represents "space". This form of representation has been borrowed from [Papa86]).

We will also look at the following three variants of $H_1$:

$$H_{10} = \begin{align*}
T_1 & : R_1(x)W_1(x) & C_1 \\
T_2 & : R_2(x) & W_2(x) & C_2
\end{align*}$$

$$H_{11} = \begin{align*}
T_1 & : R_1(x)W_1(x) & C_1 \\
T_2 & : R_2(x)W_2(x) & C_2
\end{align*}$$

$$H_{12} = \begin{align*}
T_1 & : R_1(x)W_1(x) & C_1 \\
T_2 & : R_2(x)W_2(x) & C_2
\end{align*}$$

Exchanging $R_1(x)$ and $R_2(x)$ in $H_1$ gives $H_{10}$. As these are not conflicting operations (see Eq. 2.6 in Section 2.2.4), it follows that (see Eqs. 2.12 and 2.13):

$$H_1 =_{c} H_{10}$$

However, a cyclic graph in Fig. 2.19 (stemming from the lost update $W_1(x)$ in $H_1$ and $H_{10}$) shows that:

$$H_1 \notin \text{CSR} \quad \land \quad H_{10} \notin \text{CSR}$$
$H_{11}$ basically corresponds to exchanging $R_2(x)$ and $W_1(x)$ in $H_1$, and $H_{12}$ to exchanging $W_2(x)$ and $W_1(x)$ followed by $W_2(x)$ and $R_1(x)$ in $H_{10}$. All these moves comprise conflicting operations. Thus we have:

$$H_1 
eq_c H_{11}$$
$$H_1 
eq_c H_{12}$$
$$H_{11} 
eq_c H_{12}$$

But acyclic graphs in Figs. 2.20 and 2.21 (corresponding to no lost update in $H_{11}$ or $H_{12}$) respectively show that:

$$H_{11} \in \text{CSR} \land H_{11} =_c T_1 \otimes T_2$$
$$H_{12} \in \text{CSR} \land H_{12} =_c T_2 \otimes T_1$$
Actually it is easy to see that even:

\[ H_{11} = T_1 \otimes T_2 \Rightarrow H_{11} \in S \]
\[ H_{12} = T_2 \otimes T_1 \Rightarrow H_{12} \in S \]

Second, let us proceed by respecifying \( H_2 \) as:

\[ H_2 =
\begin{align*}
T_1 & : R_1(x) & R_1(y) & C_1 \\
T_2 & : W_2(x)W_2(y) & C_2
\end{align*}
\]

We will also look at the following six variants of \( H_2 \):

\[ H_{21} =
\begin{align*}
T_1 & : R_1(x) & R_1(y) & C_1 \\
T_2 & : W_2(x)W_2(y) & C_2
\end{align*}
\]

\[ H_{22} =
\begin{align*}
T_1 & : R_1(x)R_1(y)C_1 \\
T_2 & : W_2(x)W_2(y)C_2
\end{align*}
\]

\[ H_{23} =
\begin{align*}
T_1 & : R_1(x)R_1(y)C_1 \\
T_2 & : W_2(y)W_2(x)C_2
\end{align*}
\]

\[ H_{24} =
\begin{align*}
T_1 & : R_1(x) & R_1(y) & C_1 \\
T_2 & : W_2(x)W_2(y) & C_2
\end{align*}
\]

\[ H_{25} =
\begin{align*}
T_1 & : R_1(x)R_1(y)C_1 \\
T_2 & : W_2(x)W_2(y)C_2
\end{align*}
\]

\[ H_{26} =
\begin{align*}
T_1 & : R_1(y)R_1(x)C_1 \\
T_2 & : W_2(x)W_2(y)C_2
\end{align*}
\]

\( H_{21} \) basically corresponds to exchanging \( W_2(y) \) and \( R_1(y) \) in \( H_2 \), and \( H_{24} \) to exchanging \( R_1(x) \) and \( W_2(x) \) in \( H_2 \). Both these moves comprise conflicting operations. Thus we have:

\[ H_2 \neq_c H_{21} \]
\[ H_2 \neq_c H_{24} \]
\[ H_{21} \neq_c H_{24} \]
A cyclic graph in Fig. 2.22 (stemming from the inconsistent retrievals \( R_1(x) \) and \( R_1(y) \) in \( H_2 \)) also shows that:

\[
H_2 \not\in CSR
\]

![Fig. 2.22. CSG(H_2).](image)

Exchanging \( W_2(x) \) and \( R_1(y) \) in \( H_{21} \) gives \( H_{22} \). Then exchanging \( W_2(x) \) and \( W_2(y) \) in \( H_{22} \) gives \( H_{23} \). Likewise exchanging \( R_1(x) \) and \( W_2(y) \) in \( H_{24} \) gives \( H_{25} \). Then exchanging \( R_1(x) \) and \( R_1(y) \) in \( H_{25} \) gives \( H_{26} \). As these are not conflicting operations (again see Eq. 2.6), it follows that (again see Eqs. 2.12 and 2.13):

\[
H_{21} =_c H_{22} =_c H_{23}
\]

\[
H_{24} =_c H_{25} =_c H_{26}
\]

And acyclic graphs in Figs. 2.23 and 2.24 (corresponding to no inconsistent retrievals in \( H_{21} \) or \( H_{24} \)) respectively show that:

\[
H_{21} \in CSR \land H_{21} =_c T_1 \circ T_2
\]

\[
(H_{22} \in CSR \land H_{22} =_c T_1 \circ T_2)
\]

\[
(H_{23} \in CSR \land H_{23} =_c T_1 \circ T_2)
\]

\[
H_{24} \in CSR \land H_{24} =_c T_2 \circ T_1
\]

\[
(H_{25} \in CSR \land H_{25} =_c T_2 \circ T_1)
\]

\[
(H_{26} \in CSR \land H_{26} =_c T_2 \circ T_1)
\]

![Fig. 2.23. CSG(H_2)=CSG(H_2)=CSG(H_2).](image)
Actually it is easy to see that even:

\[ H_{22} = T_1 \circ T_2 \quad \Rightarrow \quad H_{22} \in S \]

\[ (H_{23} = T_1 \circ T_2 \quad \Rightarrow \quad H_{23} \in S) \]

\[ H_{25} = T_2 \circ T_1 \quad \Rightarrow \quad H_{25} \in S \]

\[ (H_{26} = T_2 \circ T_1 \quad \Rightarrow \quad H_{26} \in S) \]

Next, we will give formal definitions of a wider set of system schedules allowing a more general kind of serializable executions with regard to interleaving.

First we need to decide - with regard to a specific item x, on which transaction \( T_j \)'s update transaction \( T_i \) is basing its computations - through a retrieval:

\( T_i \) reads-x-from \( T_j \) in \( H \) iff

- \( W_j(x) < R_i(x) \) \hspace{1cm} (Eq. 2.18)

- \( A_j \not< R_i(x) \) \hspace{1cm} (Eq. 2.19)

and

- \( \forall W_k(x) \in h \quad [W_j(x) < W_k(x) < R_i(x) \Rightarrow A_k < R_i(x)] \) \hspace{1cm} (Eq. 2.20)

Second we need to decide - with regard to a specific item x, which transaction \( T_i \)'s update the final database result is reflecting:

\( W_i(x) \) writes-x-finally in \( H \) iff

- \( W_i(x) \in h \) \hspace{1cm} (Eq. 2.21)

- \( A_i \not\in h \) \hspace{1cm} (Eq. 2.22)

and

- \( \forall W_j(x) \in h \quad [i \neq j \Rightarrow [W_j(x) < W_i(x) \vee A_j \in h]] \) \hspace{1cm} (Eq. 2.23)
Then we may formally specify the wider equivalence and serializability notions.

**View equivalent schedules** is a Binary Relation $\equiv_v$:

$$H \equiv_v H' \text{ iff}$$

- $h = h'$ \hspace{10em} (Eq. 2.24)
- $\forall x \in D, T_i, T_j$
  $$[[T_i \text{ reads-x-from } T_j \text{ in } H \land A_i \notin h \land A_j \notin h] \Rightarrow$$
  $$T_i \text{ reads-x-from } T_j \text{ in } H']$$

  and

- $\forall x \in D$
  $$W_i(x) \text{ writes-x-finally in } H \iff$$
  $$W_i(x) \text{ writes-x-finally in } H']$$ \hspace{10em} (Eq. 2.26)

**View serializable schedules** is a Set VSR:

$$H \in \text{VSR} \text{ iff}$$

- $\forall H' \text{ [H' Is-a-Prefix-of } H \Rightarrow \exists CH \in S \text{ [C(H') } \equiv_v \text{ CH]}$ \hspace{10em} (Eq. 2.27)

Eq. 2.24 says that the operational elements have to be the same, and Eqs. 2.25 and 2.26 respectively say that the reads-item-from relations between transactions and the writes-item-finally facts of transactions have to be preserved. Eq. 2.27 requires view equivalence of each possible committed projection of a prefix of the schedule to some or another serial schedule.

It is inherent in the definition that also this class of serializable schedules has the commit-projection-closed property. (See the discussion for conflict serializable schedules). So (new) view serializability corresponds to commit-projection-closedness. But old view serializability does not as it requires view equivalence of only the total schedule to some or another serial schedule. To clarify:

$$H \in \text{VSR}_n \text{ iff } H \in \text{VSR}$$

$$H \in \text{VSR}_o \text{ iff}$$

- $\exists CH \in S \text{ [H } \equiv_v \text{ CH]}$ \hspace{10em} (Eq. 2.28)
Hence $VSR_0$ has complete system schedules only as members, while $VSR_N$ (like CSR) has both complete and non-complete system schedules as members. However denoting the set of all complete system schedules $C_{CS}$, we have:

$$VSR_N = VSR$$

$$VSR_0 \supset VSR_N \cap C_{CS}$$

Before we can show how to test class membership of VSR, we have to introduce an extended version of a system schedule. Corresponding to a schedule $H$ we construct the extended schedule $H^*$ as:

$$H^* = T_w \textsc{chot} T_r$$  \hspace{1cm} (Eq. 2.29)

$T_w$ is a transaction that sets the database to a set of initial values, while $T_r$ is a transaction that senses the set of final values in a database. Thus $T_w$ writes all the items, while $T_r$ reads all the items:

$$- T_w = W_w(x_1) W_w(x_2) \ldots W_w(x_{|D|})$$  \hspace{1cm} (Eq. 2.30)

$$- T_r = R_r(x_1) R_r(x_2) \ldots R_r(x_{|D|})$$  \hspace{1cm} (Eq. 2.31)

This extended schedule construct will also be used for a class called FSR in the Appendix. The corresponding notion will also be further employed and discussed in Chapter 4.

The extended schedule construct even allows a new and simpler definition of view equivalence to be formulated (without the use of writes-x-finally):

$$H =_v H' \text{ iff}$$

$$- h = h'$$  \hspace{1cm} (Eq. 2.32)

and

$$\forall x \in D, T_i, T_j$$  \hspace{1cm} (Eq. 2.33)

$$[[T_i \text{ reads-x-from } T_j \text{ in } H^* \land A_i \notin h \lor A_j \notin h] \Rightarrow$$

$$T_i \text{ reads-x-from } T_j \text{ in } H^{**}]$$

Thus to test class membership in VSR for a given system schedule $H$, for each possible prefix $H'$ of $H$, first find the committed projection $H'' = C(H')$, second construct the extended version $H^{**}$ of $H''$, last make a directed graph-set $VSG(H^{**})$ as:
- \( V(\text{VSG}(H^*)) = \{T_w, T_r\} \cup \{T_i | t_i \in h^*\} \)  
\hspace{6.5cm} (Eq. 2.34)

- \( A(\text{VSG}(H^*)) = \)  
\hspace{6.5cm} (Eq. 2.35)
\[ \{T_w \rightarrow T_i | t_i \in h^*\} \]  
\hspace{1.5cm} (a)
\[ \cup \{T_i \rightarrow T_r | t_i \in h^*\} \]  
\hspace{1.5cm} (b)
\[ \cup \{T_j \rightarrow T_i\} \]  
\hspace{1.5cm} (c)
\[ \exists x \in D \left[ T_i \text{ reads-x-from } T_j \text{ in } H^* \right] \]  
\hspace{1.5cm} (d)
\[ \cup \{T_k \rightarrow T_j \lor T_i \rightarrow T_k\} \]  
\[ \exists x \in D \left[ T_i \text{ reads-x-from } T_j \text{ in } H^* \land T_k \text{ writes-x-also in } H^* \right] \]

(In case d) \( T_k \) is any transaction including a \( W_k(x) \) other than \( T_j \) and \( T_i \).

Then \( H \in \text{VSR} \) if and only if for each \( H^* \) there is at least one directed graph \( \text{VSG}_a(H^*) \) among the directed graph-set \( \text{VSG}(H^*) \) that is acyclic.

Cases (c) and (d) are illustrated in Figs. 2.25 and 2.26 respectively. (Fig. 2.26 shows the use of the alternative arc-pair notion. Hence the alternative arc-pair \( T_k \rightarrow T_j \lor T_i \rightarrow T_k \) means that either \( T_k \rightarrow T_j \) or \( T_i \rightarrow T_k \) but not both applies. An alternative arc-pair with \( T_k = T_w \) will be left out. An alternative arc which makes \( T_k \) either precede \( T_w \) [i.e. \( T_j = T_w \)] or succeed \( T_r \) [i.e. \( T_i = T_r \)], will also be left out. The twin arc from the corresponding arc-pair will thus be made unbroken. This happens in the more simple examples of this section. Some more complex examples are included in Sections 2.2.9 and 4.2. A discussion of the alternative arc-pair concept is given at the end of this section). Cases a) and b) respectively concern the initial setting and final sensing of the database values, and such obvious arcs will be left out in the coming examples.

![Fig. 2.25. Definition of \( T_i \text{ Reads-x-From } T_j \text{ in } H^* \) with resulting contribution of type d to \( \text{VSG}(H^*) \).]

Again note that when there is at least one acyclic \( \text{VSG}_a(H^*) \) for each \( H^* \), any topological sort of an acyclic \( \text{VSG}_a(H^*) \) for \( H^* = H \) corresponds to a serial schedule CH view equivalent to the committed projection \( C(H) \). Thus the given schedule \( H \) may once more have several view serializations.
Observe that apart from $T_w$ and $T_r$, $CSG(C(H))$ - the serialization graph of $CSR \in VSG(C(H)^*)$. It means that one specific $VSG_g(C(H)^*)$ must be acyclic for $H \in CSR$, while any general $VSG_g(C(H)^*)$ may be acyclic for $H \in VSR$. Even though VSR corresponds to some requirements on each possible committed projection of a prefix of the schedule, this leads to:

$$VSR \supset CSR$$

To clarify, also note that while (new) view serializability corresponds to employing VSG on the extended version of each possible committed projection of a prefix of the schedule, old view serializability corresponds to using VSG on the extended version of only the total schedule.

As examples for illustrating the view serializability concept, we will employ two new schedules, $H_5$ and $H_6$.

First, let us start by looking at $H_5$:

$$H_5 = \begin{align*}
T_1: & R_1(z) W_1(x) & W_1(y) & C_1 \\
T_2: & R_2(x) W_2(x) & C_2 \\
T_3: & R_3(y) W_3(y) & C_3
\end{align*}$$

From the acyclic graph in Fig. 2.27 we may deduce that:

$$H_5 \in CSR \land H_5 = _c T_1 \circ T_2 \circ T_3 = _c T_1 \circ T_3 \circ T_2$$

To check $H_5$ for class membership in VSR, we have to graph-test each possible prefix of $H_5$. We only show one of them - i.e. the graph-test of $H_5$ itself, in Fig. 2.28.
(The L associated with a vertex indicates that the corresponding transaction is live. This is a notion which is needed and explained in Section 4.2. Later an associated D will indicate a dead transaction, while an associated DL will indicate a dead transaction which is treated as live in a given context. The list of items attached to the L, D or DL shows which items the corresponding transaction updates).

From the complete set of graph-tests - with all graphs acyclic like the shown one, one may deduce that:

$$H_5 \in VSR \land H_5 = v \circ T_1 \circ T_2 \circ T_3 = v \circ T_1 \circ T_3 \circ T_2$$

This actually follows automatically from $H_5 \in CSR$ as CSR is a subclass of VSR.

From the specification we can further see that in $H_5$:

- $T_2$ reads-x-from $T_1$
- $T_3$ reads-y-from $T_1$
- $W_2(x)$ writes-x-finally
- $W_3(y)$ writes-y-finally
Second, let us proceed by looking at $H_6$:

$$
H_6 = \begin{align*}
T_1 &: W_1(x) \\
T_2 &: W_2(x)\overline{W}_2(y)W_2(v)C_2 \\
T_3 &: R_3(z)\overline{W}_3(x)\overline{W}_3(y)C_3 \\
\end{align*}
$$

From a cyclic graph in Fig. 2.29 we must conclude that:

$$
H_6 \notin \text{CSR}
$$

To check $H_6$ for class membership in VSR, once more we have to graph-test each possible prefix of $H_6$. Again we only show one of them - i.e. the graph-test of $H_6$ itself, in Fig. 2.30.

From the complete set of graph-tests - with all graphs acyclic like the shown one, one may deduce that:

$$
H_6 \in \text{VSR} \land H_6 =_{v} T_1 \sqcap T_2 \sqcap T_3 =_{v} T_2 \sqcap T_1 \sqcap T_3
$$

This indicates that CSR is a proper subclass of VSR.

From the specification we can further see that in $H_6$:

- $W_3(x)$ writes-x-finally
- $W_3(y)$ writes-y-finally
- $W_1(u)$ writes-u-finally
- $W_2(v)$ writes-v-finally
Finally, to make the differences between the VSR-class and the CSR-class more clear, let us also include a more informal description of them. We will base this discussion on the definitions of Eq. 2.6 in Section 2.2.4. Now the conflicting operations in the binary relation will be denoted:

\[ \text{R-}W_x, \text{ W-}R_x \text{ and W-W}_x \]

From Eq. 2.35 and Figs. 2.25 and 2.26 we may state that:

VSR corresponds to some combination of all relations of types

\[ \begin{align*}
&\text{W-}R_x \quad \text{[i.e. Reads-from or } x_r] \\
&\text{R-}W_x \text{ or W-W}_x \quad \text{[i.e. Writes-also or } x_w] \\
\end{align*} \]

together constituting a partial order for the committed projection of each possible prefix

[considering only "direct" relations; i.e. not transitively based]

From Eq. 2.17 and Fig. 2.18 we may state that:

CSR corresponds to (the combination of) all relations of types

\[ \begin{align*}
&\text{W-}R_x \quad \text{[i.e. Reads-from or } x_{rW}] \\
&\text{R-}W_x \quad \text{[i.e. Writes-after or } x_{rW}] \\
&\text{W-W}_x \quad \text{[i.e. Writes-before or } x_{WW}] \\
\end{align*} \]

together constituting a partial order for the committed projection of each possible prefix

[possibly considering both "direct" and "indirect" relations]

The schedules in

VSR - CSR

are those having at least one acyclic graph among the corresponding VSG-graph-set for each possible prefix even though the corresponding CSG-graph is cyclic. To make a cyclic CSG-graph acyclic within the degrees-of-freedom of the VSG-graph-set corresponds to one or more writes being moved to the right or to the left in the corresponding schedule. This is allowed in VSR by the R-W_x-or-W-W_x freedom, and not allowed in CSR by the R-W_x-and-W-W_x lack of freedom.

Imagine the occurrence of either of the two following situations in a part of the committed projection of a schedule:

\[ \begin{align*}
&W_1(x) < W_2(x) < W_3(x) \quad (a) \\
&W_1(x) < W_2(x) < W_3(x) \quad (b)
\end{align*} \]

Supposing that other parts of the schedule lead to \( T_2 - T_1 \) (non-transitively or transitively) in the CSG-graph, both the two quoted situations will necessarily close a cycle in this graph by adding \( T_1 - T_2 \) from \( W_1(x) < W_2(x) \).
Because of the existence of \( W_3(x) \), we may however move \( W_1(x) \) to the right in situation a) or \( W_2(x) \) to the left in situation b) without changing the reads-from relations of the corresponding total schedule. The results are respectively:

\[
W_2(x) (< R_4(x)) < W_1(x) < W_3(x) [< R_r(x)] \quad (a')
\]

\[
W_2(x) < W_1(x) (< R_4(x)) \quad < W_3(x) [< R_r(x)] \quad (b')
\]

With regard to the CSG-graph, either such move will transform the closing arc \( T_1 \rightarrow T_2 \) into a non-closing arc \( T_2 \rightarrow T_1 \).

And as neither such move will change the reads-from relations of the corresponding total schedule, we are just swapping schedules corresponding to the same VSG-graph-set.

For neither such move even to change the reads-from relations of any prefix of the corresponding schedule, we may not have:

\[
C_1 < C_2 < C_3 \lor C_2 < C_1 < C_3
\]

Both these two cases would give problems for any prefix not including \( C_3 \), as it is \( W_3(x) \) that saves the reads-from relations from changing under the indicated moves.

Thus we may only have:

\[
C_1 < C_3 < C_2 \lor C_2 < C_3 < C_1 \lor C_3 < C_1 < C_2 \lor C_3 < C_2 < C_1
\]

Hence for all these four cases our original schedule, including either the situation a) or the situation b), may be a member of VSR without being a member of CSR.

(This treatment does not cover all the possibilities. See Section 2.2.9 for some more cases).

However to preserve the reads-from relations of any prefix of a schedule (i.e. assuring the existence of equivalent serial schedules) in an efficient way, CSR has been defined as indicated by the description above. (Refer to Section 4.3). With reference to our original schedule including the situation a), the motives for the collective partial ordering of the given types of relations are:

- **W-R \(_x\):** Avoid \( W_2(x) \) after \( R_4(x) \)
- **R-W \(_x\):** Avoid \( W_3(x) \) between \( W_2(x) \) and \( R_4(x) \)
- **W-W \(_x\):** Avoid \( W_1(x) \) between \( W_2(x) \) and \( R_4(x) \)

Fully enforcing the second and third parts on top of the first part is definitely more than necessary with respect to correctness; i.e. achieving membership in VSR. (Again refer to Section 4.3).
2.2.7 Comments on Consistency Preservation

Observe the differences between representation and test of executions. The transaction and system schedules representing executions - i.e. \( T_i \), CH, H and C(H), are partially ordered action-sets per definition. This corresponds to that the directed graphs used for illustrating transaction and system schedules are acyclic per definition. But the directed graphs used for testing system schedules - i.e. CSG and VSG, are acyclic only in case of serializability. This corresponds to that the system schedules of analyzed executions are partially ordered transaction-sets only in case of serializability.

Acyclic serialization graphs may be assured either by making the system scheduler prevent cycles in the directed graphs, or by employing a transaction protocol that avoids cycles in these graphs. Often a combination of the two is used. For a more detailed treatment, see Section 2.3.2.

Let us now only introduce a simple and common mechanism; i.e. the 2PhaseLocking (2PL) - a transaction protocol from [Eswa76]:

2PL: No new Item may be Locked after
Some old Item has been Unlocked

1. Phase = Locking — Locked-point
2. Phase = Unlocking

Lock and unlock respectively correspond to acquire and release on a semaphore representing a single item. The rule says that the items accessed by a transaction have to be approached so that at a specific point in time (the locked-point), all items are locked together.

The applicability of different scheduling mechanisms like 2PL will be treated in depth in Section 8.1. Let us here only mention a few basic facts:

- Specifically if nothing but dynamic syntax is known about the transactions (or the database), a protocol as strong as 2PL is necessary to guarantee serializability. Dynamic syntax means that what is read or written becomes available at the time of the access.

- If something - e.g. static syntax - is known about the transactions, less stronger protocols or schedulers than 2PL are able to guarantee serializability. Static syntax means that what is read or written is known in advance of the access.

- If something - e.g. physical organisation in trees or graphs - is known about the database in addition to dynamic syntax (relating to the transactions), less stronger protocols or schedulers than 2PL are able to guarantee serializability.
- Generally if **semantics information** (i.e. functional information about the transactions or structural information about the database) and/or **integrity constraints information** (i.e. relational information about the database) is known in addition to **static syntax information** (i.e. access information about the transactions), non-serializable executions may still guarantee consistency preservation.
2.2.8 Recovery Solutions

Now we will return to the recovery problems discussed in Section 2.2.5. The goal is to define and discuss different types of system schedules corresponding to different degrees of reliability assurance. The classification employed is once more inspired by [Bern87b].

Initially, let us illustrate the solutions to the various recovery problems with informal descriptions.

A recoverable schedule is a system schedule where any active transaction may be aborted without invalidating some retrieval of a committed transaction. Thus some semantic errors are avoided.

An avoids-cascading-aborts schedule is a recoverable schedule where also any active transaction may be aborted without invalidating some retrieval of another active transaction. Thus also some recursive effects are avoided.

A strict schedule is an avoids-cascading-aborts schedule where even any active transaction may be aborted without losing some update of a committed or another active transaction. Thus even some restoration issues are avoided.

Thus correctness, efficiency and easiness in executing transactions on a database may be assured by allowing respectively only recoverable, avoids-cascading-aborts or strict schedules.

For a schedule to be recoverable, all its transactions have to be recoverable. For a transaction again to be recoverable, all its actions have to be recoverable. These restrictions concern the reads and writes but even to a greater extent the gets and puts of a transaction. See Section 2.2.2. To make a put (to the screen of a terminal) recoverable, the corresponding output message must be deferred until the transaction commits or aborts. To make a get (from the keyboard of a terminal) recoverable, the corresponding input message must be logged until the transaction commits or aborts - if automatic reexecution of failed transactions is to take place. See Section 2.4.1.

Some of the original work in this area came with [Davi73] and [Bjor73]. The basis for our presentation is taken from [Hadj88]. This was also used in [Bern87b]. An alternative approach is given in [Papa85] and used in [Papa86]. Here reliable full schedules and rollback recoverable full schedules are introduced and analyzed with regard to reliability assurance. The basis is full schedules which is an extension of system schedules employing so-called version functions. This approach illustrates an interesting and natural coupling of the recoverability theory and serializability theory. Applying single database-versions plus several log-versions of items - out of correctness reasoning with regard to reliability assurance (see Section 2.4.2), or several multi-versions of items - out of efficiency reasoning with regard to consistency preservation (see Section 2.3.2), naturally have much similarity. As usual Papadimitriou and his
coauthors are more biased towards complexity aspects, while Bernstein
and his coauthors are more biased towards pragmatically aspects.

Next, we will give formal definitions of the system schedules allowing
different types of recoverable executions with regard to faults.

First we need to decide - with regard to any item, on which
transaction $T_i$'s update transaction $T_j$ is basing its computations -
through a retrieval. (See Eqs. 2.18 to 2.20 in Section 2.2.6):

\[ T_i \text{ reads-from } T_j \text{ in } H \text{ iff } \]
\[ - \exists x \in D \left[ T_i \text{ reads-}x\text{-from } T_j \text{ in } H \right] \tag{Eq. 2.36} \]

Then we may formally specify the different recoverability notions.

**Recoverable schedules** is a Set RC:

\[ H \in \text{RC iff } \]
\[ - \forall T_i, T_j \]
\[ \left[ \left[ T_i \text{ reads-from } T_j \text{ in } H \land i \neq j \land C_i \in h \right] \Rightarrow C_j < C_i \right] \tag{Eq. 2.37} \]

This requirement is nearly, but not quite, equal to requiring:

\[ \forall x \in D, T_i, T_j \]
\[ \left[ \left[ W_j(x) < R_i(x) \land i \neq j \land C_i \in h \right] \Rightarrow [C_j < C_i \lor A_j < C_i] \right] \]

This other formulation is included to ease a comparison with the next
specification.

**Avoid-cascading-aborts schedules** is a Set ACA:

\[ H \in \text{ACA iff } \]
\[ - \forall x \in D, T_i, T_j \]
\[ \left[ \left[ T_i \text{ reads-}x\text{-from } T_j \text{ in } H \land i \neq j \right] \Rightarrow C_j < R_i(x) \right] \tag{Eq. 2.38} \]

This requirement is again nearly, but not quite, equal to requiring:

\[ \forall x \in D, T_i, T_j \]
\[ \left[ \left[ W_j(x) < R_i(x) \land i \neq j \right] \Rightarrow [C_j < R_i(x) \lor A_j < R_i(x)] \right] \]

This other formulation is once more included to ease a comparison with
the next specification.
**Strict schedules** is a Set ST:

\[ H \in ST \text{ iff } \]

- \( \forall x \in D, T_i, T_j \)
  \[ \left[ [W_j(x) < R_i(x) \land i \neq j] \rightarrow [C_j < R_i(x) \lor A_j < R_i(x)] \right] \]
  and

- \( \forall x \in D, T_i, T_j \)
  \[ \left[ [W_j(x) < W_i(x) \land i \neq j] \rightarrow [C_j < W_i(x) \lor A_j < W_i(x)] \right] \]

Eq. 2.37 says that reads-from relations between transactions imply termination-termination ordering. Eq. 2.38 says that reads-item-from relations between transactions imply termination-read ordering. And Eqs. 2.39 and 2.40 say that "reads-item-from" and "writes-item-after" relations between transactions respectively imply termination-read and termination-write ordering. The conditions for set-membership are thus gradually stronger. Hence the corresponding classes of schedules gradually solve more recovery problems, as will be seen from the coming examples.

It may also be shown that all three classes have the **commit-projection-closed property**. (See the discussions in Section 2.2.6):

\[ \]

- \( \forall H \in RC, H' \left[ H' \text{ is-a-Prefix-of } H \rightarrow C(H') \in RC \right] \quad \text{(a)} \]

- \( \forall H \in ACA, H' \left[ H' \text{ is-a-Prefix-of } H \rightarrow C(H') \in ACA \right] \quad \text{(b)} \]

- \( \forall H \in ST, H' \left[ H' \text{ is-a-Prefix-of } H \rightarrow C(H') \in ST \right] \quad \text{(c)} \]

Actually RC, ACA and ST (like CSR and VSR\(_w\)) all have both complete and non-complete system schedules as members.

As examples for illustrating the reliability notions, we will basically use the same schedules as in Section 2.2.5; i.e. \( H_4 \) and \( H_3 \) from Figs. 2.17 and 2.16 respectively. However we will rename the items to be able to combine the two schedules.

First, let us start by respecifying \( H_4 \) as:

\[
H_4 = \begin{align*}
T_1 &: R_1(y)[W_1(x)]C_1 \\
T_2 &: W_2(y)
\end{align*}
\]

The bracketed operation \( W_1(x) \) is added only to clarify the effects. The semantics of the two transactions employed are assumed to be:
\[ T_2 = y := 3 \]
\[ T_1 = x := y + 4 \]

The initial values of the two items used are supposed to be:
\[ x_s = 1 \]
\[ y_s = 2 \]

Imagine that after executing \( H_4 \) as quoted above, \( T_2 \) would like to terminate. Accepting a commit-request, \( C_2 \), represents no problem. But trying to accept an abort-request, \( A_2 \), creates major problems. The committed \( R_1(y) \) would then get invalid - after being dependent on the uncommitted \( W_2(y) \). As it is, \( T_1 \) has retrieved 3 for \( y \) and updated \( x \) to 7. Anticipating that \( T_2 \) never did execute, should have led to 2 (the initial value) being retrieved for \( y \) and \( x \) being updated to 6 in \( T_1 \). And anticipating that neither \( T_2 \) nor \( T_1 \) did execute (corresponding to "aborting" \( T_1 \) too), should have led to both \( x \) and \( y \) regaining their initial values, 1 and 2. Either way, we have to change the effects of an already committed transaction, \( T_1 \). Changing the effects of a transaction in the database is always possible - however difficult it may be. But changing the effects of a transaction with regard to the outer world may be impossible. In general it depends on the meaning of the output message of a transaction. Remember that this must not be deferred more than until the termination of the transaction. Thus we run into semantic errors. The effect is that \( H_4 \) is not a recoverable schedule. Hence \( H_4 \in RC \). To make it recoverable, we have to delay the handling of the \( C_1 \)-request until we receive either a \( C_2 \)- or \( A_2 \)-request; i.e. until \( T_2 \) terminates. This leads to the variant \( H_{41} \) of \( H_4 \) with \( H_{41} \in RC \):

\[
\begin{align*}
T_1: & \quad R_1(y)[W_1(x)] \quad C_1 \\
T_2: & \quad W_2(y) \quad C_2/A_2
\end{align*}
\]

Here, let us continue by looking at \( H_{41} \) when \( T_2 \) is about to terminate:

\[
\begin{align*}
T_1: & \quad R_1(y)[W_1(x)] \\
T_2: & \quad W_2(y)
\end{align*}
\]

The semantics of the transactions and the initial values of the items are as in the previous example.

\( T_2 \) would like to terminate. Accepting a commit-request, \( C_2 \), again represents no problem. But trying to accept an abort-request, \( A_2 \), still creates problems. The uncommitted \( R_1(y) \) would now get invalid - after being dependent on the uncommitted \( W_2(y) \). As above, \( T_1 \) has retrieved 3 for \( y \) and updated \( x \) to 7. But it should have retrieved 2 for \( y \) and updated \( x \) to 6, anticipating that \( T_2 \) never did execute. This forces an abortion (and an eventual later reexecution) of \( T_1 \) too. In general this may continue so that one abortion may lead to several abortions in sequence. Thus we run into recursive effects. The effect is that \( H_{41} \) (or \( H_{41} \)) is not an avoids-cascading-aborts schedule.
Hence \( H_{41} \notin \text{ACA} \) (but \( H_{41} \in \text{RC} \)). To make it avoids-cascading-abort, we have to delay even the handling of the \( R_1(y) \)-request until we receive either a \( C_3 \)- or \( A_2 \)-request; i.e. until \( T_2 \) terminates. This leads to the variant \( H_{42} \) of \( H_4 \) with \( H_{42} \in \text{ACA} \):

\[
H_{42} = \\
T_1: R_1(y)[W_1(x)] \\
T_2: W_2(y) \text{ } C_2/A_2
\]

Second, let us proceed by respecifying \( H_3 \) as:

\[
H_3 = \\
T_1: W_1(z)[C_1] \\
T_2: W_2(z)
\]

The semantics of the two transactions employed are this time assumed to be:

\( T_2 = z := 5 \)
\( T_1 = z := 6 \)

The initial value of the single item used is this time supposed to be:

\( z_3 = 1 \)

Again imagine that after executing \( H_3 \) as quoted above, \( T_2 \) would like to terminate. Accepting a commit-request, \( C_2 \) still represents no problem. But trying to accept an abort-request, \( A_2 \), once more creates minor problems. The \( W_1(z) \), either committed or uncommitted, could get lost - after being dependent on the uncommitted \( W_2(z) \). As it is, first \( T_2 \) has updated \( z \) from 1 (the before-image) to 5 (the after-result), then \( T_1 \) has updated \( z \) from 5 (the before-image) to 6 (the after-result). Thus aborting \( T_2 \) should not lead to the restoration of \( T_2 \)'s before-image for \( z \) (the natural isolated choice), but to the maintenance of \( T_1 \)'s after-result. Remember that \( T_1 \) has updated \( z \) after \( T_2 \).

Aborting \( T_1 \) too (imagining it has not been committed after all) should likewise not lead to the restoration of \( T_1 \)'s before-image for \( z \) (again the natural isolated choice), but to the restoration of \( T_2 \)'s before-image. Remember that \( T_2 \) has updated \( z \) before \( T_1 \) and has already been aborted. In general one may not always cancel an update of an aborting transaction in the database by exchanging the after-result with the before-image. Both these are usually saved in the log. Thus we run into restoration issues. The effect is that \( H_3 \) is not a strict schedule. Hence \( H_3 \notin \text{ST} \) (but \( H_3 \in \text{ACA} \)). To make it strict, we have to delay also the handling of the \( W_1(z) \)-request until we receive either a \( C_2 \)- or \( A_2 \)-request; i.e. until \( T_2 \) terminates. This leads to the variant \( H_{31} \) of \( H_3 \) with \( H_{31} \in \text{ST} \):

\[
H_{31} = \\
T_1: W_1(z)[C_1] \\
T_2: W_2(z) \text{ } C_2/A_2
\]
Let us finish by combining $H_3$ and $H_4$ from the previous examples into a new schedule $H_7$. Modulo commit or abort we use:

$$T_2(H_7) = T_2(H_3) \circ T_2(H_4)$$
$$T_1(H_7) = T_1(H_3) \circ T_1(H_4)$$

Thus we have (modulo commit or abort):

$$H_7 = T_2 \circ T_1$$

The resulting schedule is shown in Fig. 2.31 with an indication of class membership depending on when $T_2$ commits or aborts.

![Diagram](Image)

**Fig. 2.31.** Class membership of schedule $H_7$ as a function of Commit or Abort placement of transaction $T_2$.

This example shows an interesting point. Moving $C_2/A_2$ towards the beginning of the schedule, increases the reliability but decreases the concurrency. Moving $C_2/A_2$ towards the end of the schedule, decreases the reliability but increases the concurrency. There is a trade-off between reliability - with regard to guaranteeing recovery properties, and concurrency - with regard to allowing parallel activities.

For a fairly detailed treatment of practical mechanisms assuring recoverability, see Section 2.4.2.
2.2.9 Consistency Preservation and Reliability Assurance

Initially, the relations between the classes of schedules that we have analyzed so far (S, CSR, VSR, RC, ACA and ST), are given in Fig. 2.32. The general case covered is the so-called not-read-before-write case from Section 2.2.4. Hence we may have some blind-writes.

Fig. 2.32. How the Classes of Schedules for Consistency Preservation and Reliability Assurance are related in the General Case.

Further, the serial example $H_{70}$ is the one with the same name from Fig. 2.31 in the previous section, while the other examples will be given and analyzed in the following. We will mainly concentrate on the consistency preservation issues in this section. The reliability assurance properties may be checked as in the previous section.

First, the examples $H_{83}$ to $H_{80}$ correspond to the different variants of schedule $H_8$ depicted in Fig. 2.33.

The $H_8$-schedule is derived from the $H_7$-schedule in Fig. 2.31 by deleting the update $W_i(x)$ at the end and adding a retrieval $R_i(x)$ at the beginning.

Naturally Fig. 2.34 shows that

$H_8 \notin S$, 
while an acyclic graph in Fig. 2.35 shows that:

\[ H_8 \in CSR \land H_8 =_{c} T_2 \diamond T_1 \]

Next, the examples \( H_{g3} \) to \( H_{g0} \) correspond to the different variants of schedule \( H_g \) depicted in Fig. 2.36.

The \( H_g \)-schedule is derived from the \( H_8 \)-schedule by exchanging the retrieval \( R_1(x) \) with a retrieval \( R_1(z) \).
Here a cyclic graph in Fig. 2.37 shows that:

\[ H_9 \notin VSR \]

Last, the examples \( H_{103} \) to \( H_{100} \) correspond to the different variants of schedule \( H_{10} \) depicted in Fig. 2.38.

The \( H_{10} \)-schedule is derived from the \( H_9 \)-schedule by adding an update \( W_3(z) \) at the beginning.

Now a cyclic graph in Fig. 2.39 shows that

\[ H_{10} \notin CSR, \]

while an acyclic graph in Fig. 2.40 (and also in those corresponding to each other possible prefix of \( H_{10} \)) shows that:

\[ H_{10} \in VSR \land H_{10} =_v T_2 \otimes T_3 \otimes T_1 \]

In this case we had to choose the alternative arc \( T_2 \rightarrow T_3 \) from the alternative arc-pair

\[ T_2 \lor T_3 \lor T_1 \]

Actually for the \( H_{103} \)-prefixes we have:

\[ C(H_{10}) = H_{10} =_v T_2 \otimes T_3 \otimes T_1 \]
\[ C(H_{10} \text{ excl. } C_2) =_v T_3 \otimes T_1 \]
\[ C(H_{10} \text{ excl. } C_2 \& C_1) =_v T_3 \]
\[ C(H_{10} \text{ excl. } C_2, C_1 \& C_3) =_v - \]
Further for the $H_{102}$-, $H_{101}$- or $H_{100}$-prefixes we have:

$$C(H_{10}) = H_{10} \equiv T_2 \Join T_3 \Join T_1$$
$$C(H_{10} \text{ excl. } C_1) = \gamma T_3 \Join T_2$$
$$C(H_{10} \text{ excl. } C_1 \& C_2) = \gamma T_3$$
$$C(H_{10} \text{ excl. } C_1, C_2 \& C_3) \equiv$$

Let us even include some more examples. These will increase the understanding of the analyzed classes of schedules. They will also be referred to in Section 4.3.

First, the different variants of schedule $H_{10}$ depicted in Fig. 2.38 correspond to the examples $H_{103}$ to $H_{101}$ (but not to $H_{100}$).

The $H_{10}$-schedule is a "combination" of the $H_6$-schedule from Section 2.2.6 and the $H_7$-schedule from the previous section.

Now a cyclic graph in Fig. 2.39 shows that

$$H_{10} \notin \text{CSR},$$
while an acyclic graph in Fig. 2.40' (and also in those corresponding to each other possible prefix of $H_{10}^-$) shows that:

$$H_{10}^- \in VSR \land H_{10}^- =_v T_2 \circ T_1 \circ T_3$$

Actually for the $H_{103^{-}}$-prefixes we have:

$$C(H_{10}^-) = H_{10}^- =_v T_2 \circ T_1 \circ T_3$$
$$C(H_{10}^- \text{ excl. } C_2) =_v T_1 \circ T_3$$
$$C(H_{10}^- \text{ excl. } C_2 \& C_1) =_v T_3$$
$$C(H_{10}^- \text{ excl. } C_2, C_1 \& C_3) =_v -$
Further for the $H_{102}$-prefixes we have:

$C(H_{10}) = H_{10} \prec_T T_2 \otimes T_1 \otimes T_3$

$C(H_{10} \; \text{excl. } C_1) = v \; T_2 \otimes T_3$

$C(H_{10} \; \text{excl. } C_1 \; \text{& } C_2) = v \; T_3$

$C(H_{10} \; \text{excl. } C_1, C_2 \; \text{& } C_3) = v \; —$

And for the $H_{101}$- or $\lambda H_{100}$-prefixes we have:

$C(H_{10} \;') = H_{10} \; \prec_T T_2 \otimes T_1 \otimes T_3$

$C(H_{10} \; \text{excl. } C_1) = v \; T_2 \otimes T_3$

$C(H_{10} \; \text{excl. } C_1 \; \text{& } C_3) = v \; T_2$

$C(H_{10} \; \text{excl. } C_1, C_3 \; \text{& } C_2) = v \; —$

Next, exchange the update $W_1(x)$ with a retrieval $R_1(x)$ in schedule $H_{11}$. This results in the schedule $H_{11}$ depicted in Fig. 2.41.

![Fig. 2.41. Examples of Non-View Serializable Schedules.](image)

Here a cyclic graph in Fig. 2.42 shows that:

$H_{11} \in \text{VSR}$

Thus we will end up with examples more of the $H_9$-type. However $H_{11} \in \text{FSR}$ - see Section 4.2.3 and the Appendix; i.e. it is a consistency preserving schedule. For $H_{11} \in \text{ST}$ - i.e the $\lambda H_{100}$-case, $C_3$ must be moved in addition to $C_2$ as indicated by the dotted arrow in Fig. 2.41.

Last, refer to the schedule $H_3$ from Section 4.1. A slightly different transaction model is used in Chapter 4, so we must rephrase the schedule a little bit to match the transaction model employed in this chapter:
Trying to make an $H_{100}$-example from this schedule would force the $C_1$ to be moved to the left - out of strictness needs. But requiring that each possible committed projection of a prefix of the schedule has to be view equivalent to some or another serial schedule - corresponding to (new) view serializability as presented in this chapter, makes it impossible to move $C_1$ from its position at the end of $H_3'$ at all. However requiring that only the total schedule has to be view equivalent to some or another serial schedule - corresponding to old view serializability as mentioned in Section 2.2.6, makes it possible to move $C_1$ to a position at the end of transaction $T_1$. This results in the schedule $H_3''$:

$$
\begin{align*}
H_3'' &= T_1 : W_1(x)W_1(y)C_1 \\
T_2 : & R_2(x) \quad W_2(v)C_2 \\
T_3 : & R_3(y) \quad W_3(x)W_3(z)C_3 \\
T_4 : & W_4(x)C_4
\end{align*}
$$

Thus we have:

$$
H_3' \in \text{VSR}_N \land H_3'' \notin \text{RC}
$$

$$
H_3'' \notin \text{VSR}_N \land H_3'' \in \text{ST}
$$

Hence $H_3'$ is an example of the $H_{103}$-type. While $H_3''$ is an example more of the $H_{90}$-type. However $H_3'' \in \text{VSR}_0$; i.e. it is a consistency preserving schedule. (Fig. 4.8 in Section 4.2.3 and Fig. 4.15 in Section 4.2.5 effectively contain the VSG- and CSG-graphs of the total schedule $H_3'/H_3''$).

Looking at the $H_{10}$' the same way - i.e. trying to find an $H_{100}$-example by moving $C_1$ in addition to $C_2$, does not give any results. According to (new) view serializability, we cannot move $C_1$ at all. And according to old view serializability, we cannot move $C_1$ far enough to achieve strictness - just as far as indicated by the dotted arrow in Fig. 2.38'.

Fig. 2.42. VSG($H_1'$).
The shown examples of course depend on our interpretation of the definitions of classes VSR/CSR and RC/ACA/ST (as stated in Sections 2.2.6 and 2.2.8) being "correct".

Finally, let us look at a special case instead of the general case in Fig. 2.32. This is the so-called read-before-write case from Section 2.2.4. Hence we may have no blind-writes.

This corresponds to exchanging the last part of the definition of a transaction schedule given in Section 2.2.4 (see Eq. 2.5)

\[ [R_i(x) \in t_i \land W_i(x) \in t_i] \Rightarrow [R_i(x) <_1 W_i(x) \lor W_i(x) <_1 R_i(x)] \]

with the following:

- \[ [W_i(x) \in t_i] \Rightarrow [R_i(x) \in t_i \land R_i(x) <_1 W_i(x)] \] (Eq. 2.42)

The immediate consequence of this is that:

CSR = VSR \hspace{1cm} \text{(instead of CSR} \subset \text{VSR)}

Thus the relations given in Fig. 2.32 change into those given in Fig. 2.43.

![Fig. 2.43. How the Classes of Schedules are related in the Special Case.](image)

The more indirect consequences of this collapse again concerns the practical applicability of scheduling mechanisms, and this will be treated thoroughly in Section 4.3.
2.3 Concurrency Control

Here we will look at practical ways used to implement serializability (and to assure recoverability of a single transaction).

2.3.1 Subproblems

The main topic is interference between transactions. It is necessary to avoid bad transaction interleaving and still tolerate good transaction interleaving. Doing this in a controlled manner, means implementing one-after-the-other atomicity.

This issue raises three separate but interrelated concurrency control subproblems:

- Preserve Consistency:
  
  The integrity constraints of a database must always be satisfied. This is a genuine database system problem, and it will be fully covered.

- Avoid Deadlock:
  
  The progress of the total set of transactions must be guaranteed. This is a combined database and operating system problem, and it will be covered only partially.

- Avoid Livelock (or Starvation):
  
  The progress of each separate transaction must be guaranteed. This is a basic operating system problem, and it will be covered only briefly.
2.3.2 Policies and Mechanisms

Initially, the main responsibility for assuring consistency preservation may either be placed internally or externally:

- With the System:

  Employing a simple transaction protocol combined with an advanced system scheduler - continuously analyzing incoming actions and synthesizing serializable executions, leads to optimal concurrency at the expense of time-consuming scheduling.

- With the Transactions:

  Employing an advanced transaction protocol combined with a simple system scheduler - only checking that executing transactions observe the rules of their protocol, leads to limited concurrency at the gain of time-saving scheduling.

The responses of a system scheduler on incoming reads or writes are three-fold:

- Accepting an action means that the corresponding transaction may continue.

- Delaying an action means that the corresponding transaction has to wait some time before it may be continued.

- Rejecting an action means that the corresponding transaction has to be aborted and eventually rescheduled later.

The types of system schedulers are thus at least two-fold:

- Aggressive scheduling tries to accept an action immediately (or must necessarily reject it). Such an approach may lead to more rejects later.

- Conservative scheduling tends to delay an action initially (or may safely accept it). Such an approach will lead to less rejects later.

Further, we will introduce some actual system schedulers; i.e. real concurrency control mechanisms:

a) 2PhaseLocking (2PL):

In this mechanism each transaction dynamically acquires and releases semaphores in such a way that at a specific point in time (the locked-point) it holds all items needed. Hence this
is the mechanism already introduced in Section 2.2.7. Each transaction will consist of two phases; i.e. a growing phase where items are locked, and a shrinking phase where items are unlocked. Both deadlocks and livelocks have to be dealt with. Possible effects could be blocking or even aboritions of non-complete single transactions. The resulting serialization order will correspond to the sequence in which the transactions reach their locked-points. This method was initially presented in [Eswa76].

b) TimestampOrdering (T0):

In this mechanism the system in advance (i.e. at initiation time) assigns a timestamp to each transaction. It avoids cycles in a corresponding but non-maintained conflict serialization graph by trying to reorder any transaction conflict that would otherwise not comply with the ordering of the transaction timestamps. Possible effects could be aboritions of non-complete single transactions. The resulting serialization order will be generated a priori through the timestamp assignment process. The timestamps themselves may be generated from a physical clock or a logical counter. Such methods were effectively introduced with [Thom79] (distributed setting) and in [Reed79].

c) SerializationGraphTesting (SGT):

In this mechanism the system dynamically maintains the conflict serialization graph of class CSR from Section 2.2.6. It prevents cycles in the graph by trying to reorder any transaction conflict that would otherwise close a cycle in the graph. Possible effects could be aboritions of non-complete single transactions. The resulting serialization order will correspond to any topological sorting of the serialization graph. The serialization graph itself may be space-efficiently represented as reported in [Casa81], and terminated transactions may be time-efficiently deleted from it as reported in [Hadz86b]. Such methods were initially presented in [Bada79] (distributed setting) and with [Papa79].

d) Certifiers (C):

In this mechanism the system in retrospect (i.e. at termination time) checks whether each transaction has participated in a serializable execution or not. Possible effects could be aboritions of complete single transactions. The resulting serialization order will be calculated a posteriori through the transaction checking process. The testing may be according to the specifications of any of the three cases given above. Hence we have 2PL-, T0- and SGT-Certifiers. These methods were effectively introduced with [Thom79] (distributed setting) and in [Kung81].

Mechanism a) is protocol-heavy, while mechanisms b) - d) are scheduler-heavy.
Mechanism d) may be termed optimistic, while mechanisms a) - c) may be termed pessimistic. In some systems there will be few transaction conflicts. And if there are few transaction conflicts, there will even be few transaction aborts. In such cases mechanism d) will save more time on avoiding action checking during execution, than mechanisms a) - c) will save on carrying out transaction abortion as soon as possible.

Most comparative descriptions of such mechanisms are given for a distributed setting. See for example [Bern79a] (types a) and b)), [Bern80a] (type b)), [Bern81b] (≠ types a) - d)), [Bern82] (types a) - d)) and [Koh181] (also covering recovery mechanisms).

The same applies to many comparative performance analyses. See for example the simulation results in [Bada81]. [Sevc83] also gives some analytic results. [Tay85] reports their own results - on locking specifically, while [Tay84] surveys results from others - again on locking specifically. [Lin82] focuses on mixtures of read-only and read-write transactions - once more with locking. [Morr85] compares several certifier methods versus locking methods.

One very interesting possibility is the option to dissect concurrency control on different types of conflicting operations (see Section 2.2.4):

One may separate RW-concurrency control from WW-concurrency control. Hence it is possible to synchronize reads vs. writes (& writes vs. reads) separately and differently from writes vs. writes - provided an arrangement is added to make the two resulting serialization orders consistent. Any of the four mechanisms given above may be used on each of the two different synchronization aspects - giving a lot of mixed cases.

This issue was initially touched in [Bern80a] and treated in full in [Bern81b]. There also appeared a new synchronization method only applicable to WW-concurrency control out of this. It is the ThomasWriteRule - originally stemming from [Thom79].

Another very interesting aspect is the possibility to keep several non-identical versions of each item in time. These issues were effectively introduced with [Thom79] (distributed setting) and in [Reed79]. Both [Bern83a] and [Papa84] have later contributed to a general multiversion serializability theory. [Bern83a] even includes some algorithmic descriptions, while [Hadz86a] contains some complexity discussions. With regard to specific examples; see [Stea81] for a 2-version 2PL based method; see [Thom79] (distributed variant), [Reed79] and [Bern83a] for T0 based methods; and see [Hadz86a] for an SGT based method. Also see [Baye80a] (distributed variant) and [Baye80b] for 2-version SGT-Certifier & 2PL mixed methods. A performance analysis comparing several multiversion methods may be found in [Care84]. This even includes the so-called hierarchical locking to be mentioned later.
Next, some of the involved aspects of concurrency control can be illustrated using 2PhaseLocking as a case. This will also pinpoint some of the possibilities and limitations of this specific mechanism itself.

As mentioned in Section 2.2.7, locking and unlocking items respectively correspond to acquiring and releasing semaphores.

It is quite normal to distinguish between different kinds of locks, like:

- X:
  
  This exclusive-lock type may be used to correctly solve the lost update + inconsistent retrievals problems (and the update dependent on uncommitted update + retrieval dependent on uncommitted update problems). See Section 2.2.5.

- S:
  
  This shared-lock type may be used together with X-locks to more efficiently solve the inconsistent retrievals problem (and the retrieval dependent on uncommitted update problem).

- U:
  
  This update-lock type may be used instead of X-locks to increase the efficiency - by decreasing single blocking, and instead of S-locks to avoid some deadlocks - by preventing multiple blocking.

The 2PL transaction protocol actually requires that each transaction which locks and unlocks items, is:

- Well-Formed:

  A transaction has to lock an item before using it, and unlock an item after using it. A transaction cannot lock an item which it already has locked, or unlock an item which it already has unlocked.

- Legal:

  A transaction has to obey a compatibility matrix expressing which types of locks may and may not be set on an item at the same time. See Fig. 2.44.

- 2PhaseLocked:

  A transaction cannot lock any further items after some initial item has been unlocked. Hence, first lock all items needed, then unlock all items used.
A set of items which is changing continuously and accessed by content (see Section 2.1.2), leads to the problem of so-called phantom items. This may be solved through predicate locking (advocated by [Eswa76]), or by locking indices of items instead of or in addition to locking items themselves. Another interesting discussion of this topic may be found in [BernB3c].

Locking both indices of items and items themselves extends naturally to locking in any number of levels. Such multigranularity systems were introduced by the intention locking of [Gray75]. It has later been generalized from locking in hierarchies of data types to locking in networks of data types. Thus we have logical tree/graph locking. [KortB3] further explores the possibilities and limitations of different locking modes - i.e. combining basic locks like X and S through basic operations like union and upgrading, both for singlegranularity and multigranularity systems.

The notation used in the coming examples has to be explained:

- \( LX_i(y) \) : Request an X-Lock on Item \( y \) for Transaction \( i \)
- \( LS_i(y) \) : Request an S-Lock on Item \( y \) for Transaction \( i \)
- \( LU_i(y) \) : Request a U-Lock on Item \( y \) for Transaction \( i \)
- \( * \) : Grant the Requested X/S/U-Lock
- \( UL_i(y) \) : Release (Unlock) Acquired Lock on Item \( y \) for Transaction \( i \)
- \( \text{Does/Must happen Before/After} \)
- \( \text{Wait through Blocking} \)

First, two illustrations of a lost update solution and an inconsistent retrievals solution are given in Figs. 2.45 and 2.46 respectively.
Fig. 2.45. Avoidance, using X-locks, of the Lost Update problem (two Read-Write transactions accessing one item).

Fig. 2.46. Avoidance, using X- & S-locks, of the Inconsistent Retreivals problem (one Write-Only transaction and one Read-Only transaction accessing two items).
Second, an illustration of the **efficiency-gains** from using U-locks instead of X-locks is given by Figs. 2.47 and 2.48.

![Diagram of inefficient solution with X- & S-locks](image)

**Fig. 2.47.** Inefficient solution with X- & S-locks to allow simultaneous Read-Write and Read-Only access to one item (a non-problem).

![Diagram of efficient solution with (upgraded) U- & S-locks](image)

**Fig. 2.48.** Efficient solution with (upgraded) U- & S-locks to allow simultaneous Read-Write and Read-Only access to one item (a non-problem).
Third, an illustration of the **necessity** of the 2PL-*requirement* is given in Fig. 2.49.

![Diagram](image)

**Fig. 2.49.** Incorrect solution with a Non-2PL transaction to the Inconsistent Retrievals problem. (T₁ ought to be aborted.)

Fourth, an illustration of the **possibility** of a *deadlock-occurrence* is given in Fig. 2.50.

![Diagram](image)

**Fig. 2.50.** Deadlock possibility with X-locks on two items. (T₁ or T₂ must be aborted.)
Fifth, an illustration of the deadlock-reductions from using U-locks instead of S-locks is given by Figs. 2.51 and 2.52.

Fig. 2.51. Deadlock possibility with S-locks upgraded to X-locks on one item. (T₁ or T₂ must be aborted).

Fig. 2.52. Deadlock impossibility with U-locks upgraded to X-locks on one item.
[Holt72] theoretically showed that each deadlock corresponds to a cycle in a so-called WaitForGraph. Such a graph relates which transactions are waiting (through blocking) for which other transactions, and [Gray78] introduced practical ways of representing and manipulating these graphs. [Yann82b] further discusses freedom of deadlocks totally.

The general deadlock problem may be approached in a lot of different ways:

- **Deadlock Avoidance:**

  These methods make cyclic transaction blocking impossible. Hence transaction abortion will never be necessary because of this.

  i) Use linearization to order the items, and force the transactions to do locking according to this

  ii) Lock all items needed together with one lock

  This method requires a predeclaration of the items to be used.

- **Deadlock Prevention:**

  These methods act when cyclic transaction blocking might/will occur. The transaction to be aborted is fixed; i.e. the one possibly closing the WaitForGraph cycle.

  i) Use timestamps to order the transactions, and expect the transactions to allow blocking according to this

  Such a method was presented in [Rose78] (distributed setting), both with a non-preemptive version (WaitDie) and a preemptive version (WoundWait).

  ii) Check the WaitForGraph each time a blocking relation is to be added

- **Deadlock Detection:**

  These methods check whether cyclic transaction blocking might have/will have occurred. The transaction to be aborted is selected - among those possibly constituting a WaitForGraph cycle.

  i) Interpret timeouts correctly/wrongly as deadlock indications

  ii) Check the WaitForGraph periodically (after several blocking relations could have been added)

  Such methods (all in distributed settings) were presented in [Gray78], [Mena79], [Ober82] and [Chan83].
The special **livelock problem** has at least two **subcases**:

- When a transaction is **requesting a lock**, it may have to wait due to some locks granted to other transactions. If the selection algorithm with regard to whose lock-requests are to be granted next, is not fair, a transaction may be blocked for ever. A simple and fair algorithm is to grant the lock-requests of the transactions that have waited the longest; i.e. a FIFO-rule.

- When a transaction is **involved** in a **deadlock**, it may have to be aborted. If the selection algorithm with regard to which transaction is to be aborted, is not fair, the transaction may be cyclically rescheduled. A simple and fair algorithm is to abort the transaction that has existed the shortest; i.e. a LIFO-rule.

[Gray81b] contains a more theoretical discussion of both these aspects; i.e. livelocking in connection with deadlocking and blocking. [Blas79] contains a more pragmatically discussion of the so-called thrash-locking - a topic related to some recursive waiting effects of blocking.

Finally, the **2PhaseLocking-requirement** is enough to solve all the indicated **concurrency control problems**. However by extending it into a LockUntilEnd-requirement - i.e. hold all the acquired locks all until the transaction commits and not only until all locks to be requested have been granted, the mechanism will also solve the single transaction **recovery problems** indicated.

Let us thus analyze a plausible set of **combinations** with an **exclusive-lock** and a **shared-lock**, see Fig. 2.53. A **short lock** is only held while the corresponding action is carried out, while a **long lock** is held from the corresponding action is carried out until the transaction itself terminates.

![Figure 2.53: Used combinations of Exclusive-Locks and Shared-Locks.](image)

First, Fig. 2.54 illustrates a problem possible in an A) case or in an A) case extended with long S-locks as here. The **retrieval** \( R_1(y) \) or the **update** \( W_1(y) \) gets dependent on the **uncommitted update** \( W_2(y) \). An invalid retrieval or a lost update results at the abortion \( A_2 \). As transaction \( T_1 \) is already committed when the invalidation or loss occurs, the troubles may be unrepairable.
Fig. 2.54. The retrieval/update of $T_1$ gets invalid/lost at the abortion of $T_2$.
We have non-recoverability of the write of $T_2$
with regard to the read/write of $T_1$.
(The schedule is Non-Recoverable/Non-Strict).

Also in Fig. 2.55 - still illustrating the problem possible in an A) case or in an A) case extended with long S-locks as here, we effectively get the same results. But as transaction $T_1$ does not terminate until transaction $T_2$ terminates, the troubles may be repairable.

Fig. 2.55. The retrieval/update of $T_1$ still gets invalid/lost at the abortion of $T_2$.
We have still non-recoverability of the write of $T_2$
with regard to the read/write of $T_1$.
(The schedule is now Non-Avoids-Cascading-Aborts/Non-Strict).
It is only in Fig. 2.56 - now illustrating the problem’s solution with a B) case or with a C) case as here, that we avoid the invalid retrieval or the lost update.

The recoverability of a write with regard to another write or read means that the corresponding writing transaction may later be aborted without side-effects. Eventual side-effects of the non-recoverability of a write could be enforced abortions of other (writing or reading) transactions as well, see the discussions in Section 2.2.8. The recoverability itself of a write requires that X-locks are held until the transaction commits.

Second, Fig. 2.57 illustrates a problem possible in an A) case or in a B) case as here. The retrieval $R_2(y)$ finds different values in the two executions.

But in Fig. 2.58 - now illustrating the problem’s solution with a C) case, the retrieval finds equal values in the two executions.

The repeatability of a read with regard to another write means that the corresponding reading transaction may later be rescheduled (after an abortion) with identical results. Eventual non-identical results from the non-repeatability of a read imply that separate parts of the output of the transaction could correspond to separate retrievals not reflecting a single consistent database state. In this case the output message(s) of the reading transaction should be deferrable until the transaction terminates. The repeatability itself of a read requires that S-locks are held until the transaction commits.
Fig. 2.57. The value retrieved by T₁ is different from the value initially retrieved by it. We have non-repeatability of the read of T₁.

Fig. 2.58. The value retrieved by T₁ is now equal to the value initially retrieved by it. We now have repeatability of the read of T₁.
To summarize, the concurrency control and recovery properties guaranteed at the different levels are shown in Table 2.2.

Table 2.2. Properties guaranteed by all transactions executing at the indicated level as a minimum.

<table>
<thead>
<tr>
<th>Level</th>
<th>Concurrency Control &amp; Recovery Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Recoverability of Writes (with regard to Writes &amp; Reads)</td>
</tr>
<tr>
<td></td>
<td>⪰: C_a ⊃ VSR &amp; C_a ⊆ RC</td>
</tr>
<tr>
<td>C</td>
<td>⪰: C_e ⊃ VSR &amp; C_e ⊆ ST</td>
</tr>
</tbody>
</table>

C_a indicates: The class of schedules corresponding to level a

The mentioned class-relationships are only rough indications of how the classes are related. (A ⪰ is supposed to mean more inclusive than a ⪰). Here we are only concerned with the basic problems introduced in Section 2.2.5 and this section, and the mentioned class-relationships are thus not meant to have general validity. A formally correct and detailed analysis of classes C_a - C_e will be included in Section 9.4.1. See also Section 5.3. (Note that class C_a comprises all schedules in which each transaction executes at least level a).

The 2PL class referred to corresponds to the set of schedules achievable with a pure 2PL-mechanism. We will return to the relationship between this class and other classes in Chapter 4. All the other classes referred to were specified in Sections 2.2.6 and 2.2.8.

Thus we have to go from 2PhaseLocking to LockUntilEnd (LUE) to avoid both the concurrency control and recovery problems.

With regard to correctness: one may always lock more than what is actually required - i.e. a collection of items instead of a single item, longer - i.e. until the termination-point instead of until the locked-point, and stronger - i.e. exclusive instead of shared. Further there are no problems either with upgrading locks (from S or U to X, which is necessary when an item read, also has to be written) or downgrading locks (from X or U to S, which is possible when an item read, actually does not have to be written).

[Ries77] and [Ries79] present some simulation results concerning the interesting aspect of an optimal lock-unit size. This even relates to the hierarchical locking mentioned earlier.
2.4 Recovery

Now we will look at practical ways used to implement recoverability of single and multiple transactions.

2.4.1 Subcategories

The main topic is failures of transactions. It is necessary to avoid bad transaction faults and still tolerate good transaction faults. Doing this in a controlled manner, means implementing all-or-nothing atomicity.

This issue concerns three recovery subcategories, which are gradually more severe:

i) Transaction-Failures:

This concerns error conditions relating to a specific transaction without any loss of storage. Such error conditions may be detected/ enforced either by the transaction - e.g. the transaction finds a wrong value when accessing an item, or by the system - e.g. the system selects the transaction as victim in a deadlock resolution. The specific transaction must be aborted in both these situations. In some cases it is not immediately reexecutable - e.g. in the wrong value situation, while in other cases it is immediately reexecutable - e.g. in the victim selection situation. Immediate rescheduling should be carried out automatically by the system, while non-immediate rescheduling would be carried out manually by the user. Only the specific transaction can be affected.

ii) System-Failures:

This concerns a loss of volatile storage like primary memory. One reason could be a power failure. The effect is considered to be a loss of the complete buffer for database and log items (see Section 2.2.3). Thus all active transactions will be affected.

iii) Media-Failures:

This concerns a loss of non-volatile storage like secondary memory. One reason could be a single disk crash. The effect is considered to be a loss of the complete current database only (see the next section). Thus all transactions will be affected, but the damage is fully repairable. The described case is a good one.

This also concerns a loss of so-called stable storage; i.e. more secure storage hierarchically implemented from several non-volatile storage components (see [Lamp81a]). One reason could be multiple disk crashes. The effect is considered to be a loss of the complete current database and log plus eventually a loss of one or more old dumps of complete
databases and one or more old versions of logs (see the next section). Thus all transactions will be affected, and the damage is not fully repairable. The considered case is a bad one.

Note that for category i), the previous section partially dealt with how to minimize the effects on other transactions of an eventual abortion of some transaction, while the next section will deal with how to carry out the actual abortion of a single transaction.

Observe that for categories ii) and iii) we anticipate that a loss of some type of storage is total and never partial. The memory of a specific kind is either functioning perfectly or not functioning at all for a given system. Thus all or none of the corresponding transactions can be affected. But out of these multiple transactions, some must be aborted, while others may be terminated in a normal way.
2.4.2 Tools and Mechanisms

An early general analysis of the aspects involved in recovery is given in [Rand78], while an early general survey of recovery mechanisms used is given in [Verh78]. A thorough discussion of the recovery part of a complete hypothetical system may be found in [Lind80a] (covering both centralized and distributed topics), and the same may be found in [Gray81c] for a complete existing system (System R).

The different kinds of memory involved in recovery mechanisms are illustrated in Fig. 2.59. It represents a slightly different perspective than that of Fig. 2.11 in Section 2.2.3.

![Diagram of recovery-related storage]

Fig. 2.59. Recovery-related Storage.

Initially, let us look at the extra storage that may be used in connection with recovery (besides that containing the current system database itself).

- System Cache (containing Buffer memory):

  Size considerations imply that the database basically has to be stored in secondary non-volatile memory. But speed considerations imply that parts of the database additionally has to be stored in primary volatile memory.

  Before an item can be read, its value has to be fetched from the database to the cache - if it is not already there. And after an item is written, its value has to be flushed from the cache to the database - but this does not have to occur immediately. [Effo84] discusses such cache administration in detail. The current database state is thus represented in a combination of the non-volatile database and the volatile cache.

- System Log:

  Security considerations lead to a need to record descriptions of the database operations sequentially in time. The logged descriptions will generally make it possible both to undo a database operation (arrange as if it did not happen at all) and to redo a database operation (arrange as if it actually did happen). This occurs at the level of a single item access.
Log-records are buffered like database-records, but log-records are only written and hopefully never read. The corresponding flushing ought to be optimized as many log-records usually fit in one log-page. This normally leads to a deferred log-flushing, but a forced log-flushing may be needed in specific situations.

The correctness aspects of the database and log writing may be formulated in a LogWriteAhead protocol.

Undo-rule: (per Action)

Any item update must be reflected in the non-volatile log before its write can be flushed to the non-volatile database

Redo-rule: (per Transaction)

All item updates must be reflected in the non-volatile log or database before a transaction can be committed

Both these rules may require log-flushing.

It is very important that a database and its log have independent failure modes; e.g. requiring separate storage components.

- System Backup (containing either Dump or Copy memory):

Security considerations also lead to a need to periodically file the database state by dumping the current database values on another storage medium - and/or continuously mirror the database state by copying all current database changes to separate storage media. The archive writing of the dumping process results in several non-identical occurrences of the database, while the duplex writing of the copying process maintains several identical occurrences of the database. Both these occur at the level of multiple item values.

It is possible to have backup versions of the log too. Dumping a log is usually carried out to empty a (nearly) full log. Thus separate versions of the log contain separate parts of the complete log.

Again it is very important that the current database plus log and their backups have independent failure modes.

Further, let us introduce some actual recovery mechanisms used for the different types of failures from the previous section:

i) Transaction-Failures:

Such a failure requires an implicit system abortion of a single transaction. As for an explicit user abortion one
employs the (volatile and non-volatile) current database and log. The specific transaction cannot have a commit-record in the log, and it must thus be aborted (undone).

a) Undo

The algorithm basically traverses the log backwards undoing any operation that corresponds to the aborting transaction and that may have been carried out.

ii) System-Failures:

Such a failure requires an (emergency) restart of a set of multiple transactions. One employs the (non-volatile) current database and log. Those transactions that have no commit-record in the log, must be aborted (undone). And those transactions that have a commit-record in the log, may be terminated in the normal way (redone).

a) Undo/Redo

This algorithm basically traverses the log backwards undoing any operation that corresponds to a non-committed transaction and that may have been carried out—and traverses the log forwards redoing any operation that corresponds to a committed transaction and that may not have been carried out.

In this general variant any corresponding database-flushing is allowed to take place prior to a transaction-commit, and all corresponding database-flushing is not forced to take place at the transaction-commit. Hence both the undo- and redo-parts will be needed. For examples, see [Gray78] (general discussion) and [Gray81c] (System R).

b) Undo/No-Redo

In this variant of the algorithm any corresponding database-flushing is still allowed to take place prior to a transaction-commit, but all corresponding database-flushing is now forced to take place at the transaction-commit. Hence the redo-part will never be needed. For an example, see [Chan82] (DDM).

c) No-Undo/Redo

In this variant of the algorithm no corresponding database-flushing is now allowed to take place prior to a transaction-commit, but all corresponding database-flushing is still not forced to take place at the transaction-commit. Hence the undo-part will never be needed. For examples, see [Lamp81a] (discussing careful-replacement) and [Ston79] (Ingres).

d) No-Undo/No-Redo

In this variant of the algorithm no corresponding database-flushing is now allowed to take place prior to a transaction-commit, and all corresponding database-flushing
is now forced to take place at the transaction-commit. Hence neither the undo- nor the redo-part will ever be needed. For an example, see [Lori77] (discussing shadowing).

iii) Media-Failures:
Such a failure requires a reset (cold start) of a set of multiple transactions. We will only discuss the variant with archive writing but without duplex writing. In the good case one employs the latest (non-volatile) backup database and all necessary log versions. While in the bad case one employs the latest surviving (non-volatile) backup database and all necessary surviving log versions. Those transactions that have a commit-record in the remaining part of the log, may be terminated in the normal way (redone).

a) Redo
The algorithm basically traverses the log forwards redoing any operation that corresponds to a committed transaction and that may not have been carried out.

In the good case the database state existing at the time of failure is actually reproducible. It reflects all of the committed transactions at that time. But in the bad case only a database state possibly neither existing at the time of failure nor at any time before that may be producible. It possibly reflects only some of the committed transactions.

We have followed the grouping from [Haer83] and [Bern83b], which is based on algorithmic effects. Another comparative description of such mechanisms is given in [Kohl81] (also covering concurrency control mechanisms). A comparative performance analysis may be found in [Reut84].

Finally, let us indicate ways of increasing the efficiency of some recovery mechanisms.

- Checkpointing:
A database-flushing corresponding to all the item updates whose writes have not been flushed already, will make the current database state completely reflected in the non-volatile database - and not only in a combination of the non-volatile database and the volatile cache. Periodically carrying out such a recording of the database state all in non-volatile storage, will imply that less redo is needed in the Undo/Redo and No-Undo/Redo algorithms. Even recording the transaction states with regard to executional progress in non-volatile storage, will further imply that less searching is needed to clarify which transactions to redo and which to undo.
- Dump Checkpointing:

Applying the same ideas to the creation of dump backups, will result in an incremental dumping. This will for example make the inactive periods implied by the dumping process much shorter.
3 Distributed Database Systems

In this chapter we present our view of distributed systems and transaction processing in distributed databases. The main issues are again consistency preservation and reliability assurance in the presence of even true interference and partial failures. Hence once more concurrency control and recovery are the main subjects to be treated. A general discussion of distributed systems may be found in [Lamp81b], while [Ceri84] and [Bray82] deal with distributed databases specifically. An overall treatment of transaction processing in such systems may be found in [Date83] and [Ullm82] partially, while [Bern87b] and partially [Papa86] cover concurrency control and recovery in these systems in detail.
3.1 Theme and Theory

This section contains an introduction to and elaboration of the basic concepts and notions of this chapter.

3.1.1 Distributed System Definition and Characterization

It is possible to employ the following informal description of a distributed database system:

A distributed database system comprises a collection of centralized database systems (sites) connected together with communication channels (links). The computer locations store a collection of datasets which belong to one logically unique system, but which are spread over several physically dispersed systems. The computer network allows a single site to access data at multiple sites, and allows multiple sites to access data at a single site.

However, it is more interesting to clarify the essential differences between distributed and centralized systems per se:

A distributed system is a system with lack of instant signal observability, see [LeLa81]. This means that an event being produced in some part of the system — e.g. the introduction, change or removal of a specific data-item, may not be immediately observed in all parts of the system. In contrast, in a centralized system event production and event observation coincide. The effect is that separate parts of the system (associated with separate sites) may have non-consistent perceptions of the system time — e.g. which events happen before/after other events, the system status — e.g. which parts/sites are functioning/not functioning, and the system state — e.g. which data-values correspond to the different real world objects being represented.

This lack of instant signal observability or non-consistent perception of time, status and state is due to the differences in operating speed between the communication and storage media:

- \(\text{Time(Send/Receive-on-Link)} \gg \text{Time(Read/Write-at-Site)}\) \ (Eq. 3.1)

Currently, non-shared memory and clock systems are examples of configurations that have poor signal observability and non-common time, status and state perception. While shared memory and clock systems have excellent signal observability and common time, status and state perception.
3.1.2 System Architecture and Assumptions

A distributed database systems model corresponding to the definition and characterization of the previous section is given in Fig. 3.1. (See also the centralized database systems model in Section 2.2.3).

Fig. 3.1. High-Level Model of a Distributed Database System.

We would like to make some specific distinctions with regard to terms. These totally reflect our own opinion.

A collection of data-items from several separate sites constitutes a global database:

\[
\text{Global Database} = \bigcup_{i=1}^{S} \text{Local Database}_i
\]

\((S\) being the number of sites\)

The addition of connecting links between the separate sites results in a system database:

\[
\text{System Database} = \text{Global Database} + \text{Communication Facility}
\]
A decentralized database system includes a global database plus a communication facility being built separately and in sequence. Hence, first several isolated centralized database systems with local databases are constructed and used, then a computer network is added on top to connect the systems.

While a distributed database system includes a global database and a communication facility being built collectively and in parallel. Hence the dispersed-storage and inter-communication of data-items are designed as a total system.

We ought to mention some of the basic assumptions underlying much of the discussions concerning distributed database systems:

- A homogeneous system is assumed - in contrast to a heterogeneous system. This means that the sites are supposed to be similar. The advantage of this is manageable subproblems. Hence, first the issues associated with even homogeneous parts are solved, then the added issues from heterogeneity may be addressed. (See for example [Spac80]).

- A long-haul network is assumed - in contrast to a local area network. This means that the links are supposed to be slow. The effect of this is a resulting goal of minimization of the number and volume of messages on the communication media - and not a minimization of the number and volume of accesses in the storage media. (However see a discussion in [Nygå86] about the current trend and associated consequences of the development of the relative speeds of communication and storage media).
3.1.3 Distributed Database System Objectives and Advantages

First, let us investigate the sensible goals of a distributed database system:

- Implement atomic transactions with regard to concurrency control and recovery. This was the basic theme in the previous chapter about centralized database systems, and it will also be the main theme in this chapter. (Atomic transactions’ implementation should also address integrity aspects and security aspects, see for example [Buss82]).

- Optimize global access with regard to the use of data-items from multiple sites - on top of optimizing local access (from centralized database systems) with regard to the use of data-items in a single site.

- Allow data fragmentation and data replication with regard to user- and system-data. Fragmentation means dividing one logical data-item into several smaller data-item parts, while replication means making several physical data-item(part) copies for one data-item(part). Data location then concerns where to place each logical/physical data-item(part). In the relational data model fragmentation may be horizontal - corresponding to a selection, vertical - corresponding to a projection, or mixed - corresponding to a combination thereof. Any replication is naturally counter-effective to the reduced redundancy goal in centralized database systems. The data location may actually lead to any subset of the global database being stored in each local database.

- Assure global data transparency with regard to data fragmentation, data replication and data location between sites - on top of assuring local data independence (from centralized database systems) with regard to data storage within a site. The added result is a centralized system view even in the distributed system.

- Accept local site autonomy with regard to user- and system-processes. This means granting the separate sites rights to decide in local matters. Any local autonomy is naturally counter-effective to the unique control goal in centralized database systems. The result is a partnership among independent but cooperating centralized systems - instead of a uniquely controlled distributed system. This is counter-effective to most of the other objectives listed above, and it will be another important topic in this chapter.

- Decentralize data occurrence and process control as much as possible, see for example [Jens81]. Emphasizing a distribution of the occurrence of user- and system-data demands high fragmentation. Increasing the retrievals in a system will further demand high replication, while increasing the updates in a system will instead demand little replication. Emphasizing a distribution of the control of user- and system-processes demands high local autonomy. (For another early discussion of the design and administration of a distributed database system - especially with regard to who does what locally or globally, see [Gros80]).
Second, let us investigate the possible advantages of a distributed database system versus a centralized database system:

- **Combination effect** of connecting several initially isolated local databases. This primarily concerns decentralized database systems. One gets:
  - **Accessibility** for a specific user to data at his original site plus data at all other sites.

- **Localization effect** of dividing one initially large local database into several parts placed at different sites. This concerns data fragmentation and data location. One achieves:
  - **Adaptation** to organizational structure; i.e. option to place data according to ownership.
  - **Efficient** and **flexible access** for a single transaction to a single data-item; i.e. option to place (and later move) data according to closeness (or later change) of use.
  - **Fast access** for a single or multiple transactions to multiple data-items. This means possibilities for parallel reads and/or writes on different items; i.e. allowing parallelism within a single query-part or a single updater-part of a transaction or between multiple query-parts and/or multiple updater-parts of separate transactions.
  - **Continued access** despite crash of sites. Hence a specific transaction only retrieving/updating items at sites still functioning, does not have to halt. This equals the survival of errors of some sites - or some sets of data-items. We have achieved increased reliability of the database system - in the meaning of a stepwise degradation of the system functioning.

- **Duplication effect** of making several copies of one or more parts of an initially local or global database and spreading these over several sites. This concerns data replication and data location. One achieves:
  - **Fast access** for multiple transactions to multiple copies of a single data-item. This means possibilities for parallel reads and a write eventually on different copies of an item; i.e. allowing parallelism between multiple query-parts and a single updater-part eventually of separate transactions.
  - **Continued access** despite more crash of sites. Hence a specific transaction only retrieving/ updating items having at least one copy each at sites still functioning, does not have to halt. This equals the survival of more errors of some sites - or some sets of copies of data-items. We have achieved increased availability of the database system.
- General effect of dealing with units of some specific prices and sizes. This concerns:

- **Economy**: i.e. one may get a cheaper total (or partial) configuration.

- **Capacity**: i.e. a final configuration with a larger total result is possible.

- **Growth**: i.e. an evolving configuration through smaller intermediate steps is possible - in the meaning of a stepwise escalation of the system capacity.
3.1.4 Distributed Database System Problems or Issues

Based on the goal discussion in the previous section (and on Sections 3.1.1 and 3.1.2), we will look at what kind of problems or issues that a distributed database system presents us with. (See [Roth77b] for one of the earlier source-materials analyzing this).

- Signal Observability:

  - Implementing global system time may be obtained through the logical clocks concept of [Lamp78].
  
  - Implementing global system status may be obtained through the sites crash/repair monitoring and broadcasting notion of [Walt82].
  
  - Implementing global system state have to be obtained through updates distribution.

In a centralized database system a collection of coexisting updates have to be carried out on all corresponding items, at the unique site. Introducing data fragmentation leads to a mandatory propagation of updates to all corresponding items at all involved sites. Then adding data replication further leads to a mandatory propagation of updates to all occurring copies of all corresponding items at all involved sites. With both these aspects of a distributed database system spooling-sites may be used to overcome the possibility of the sender and receiver being up/down at different times.

- Atomic Transactions:

  - Implementing concurrency control is an absolute requirement.

On top of the usual one-after-the-other aspects of centralized database systems, data fragmentation without data replication only adds efficiency considerations - stemming from the existence of separate sites with different logical items. A single conflict between transactions - i.e. concerning one unique item, will be detected as before. But violation of serializability requires multiple conflicts concerning several different items, and this may be detected much later than before due to the possible need of inter-site communication. Thus more work may be wasted. (See Section 3.2.1).

But data fragmentation plus data replication adds further correctness considerations on top of those of centralized database systems - stemming from the existence of separate sites with different physical item copies of a single logical item. Even a single conflict concerning one unique logical item could possibly remain undetected as it does not necessarily manifest itself at a specific physical item copy. To avoid this, a common physical control-copy or fixed separate control-point is needed for each logical item for concurrency control purposes. (See Section 3.3.1).
- Implementing recovery is an absolute requirement.

On top of the usual all-or-nothing aspects of centralized database systems, data fragmentation without data replication also adds correctness considerations - stemming from the existence of separate sites with different logical items. A single site may either function or not function as before. But among multiple sites one or more may function, while one or more others may not function. Thus instead of having a total failure of a system or not, we may have partial failures in the system. This leads to the requirement of the so-called atomic commitment to achieve recoverability. Here it means that before an operation inducing changes in several sites may be carried out, all the involved sites have to be inquired with regard to their ability to perform the corresponding actions. All of them have to respond positively before an actual decision to go through with the operation may be taken. (See Section 3.2.2).

And data fragmentation plus data replication adds even further correctness considerations on top of those of centralized database systems - stemming from the existence of separate sites with different physical item copies of a single logical item. The crash or repair of a single site including a specific physical item copy may be detected at different times by separate sites due to the need of attempted inter-site communication. Thus the perception of one unique logical item may vary from site to site. To avoid the consequences of this, a common failure/non-failure view of the physical copies is needed for each logical item for recovery purposes. (See Section 3.3.2).

- Global Access:

- Optimizing query transactions is both a possibility and a necessity. There is a choice - from data fragmentation, data replication and data location - with regard to where to read old data-item values and where to compute new data-item values from old data-item values, and the time used on different approaches varies considerably. This is a plan topic; i.e. it concerns efficient data manipulation.

Some results from simulation experiments may be found in [Epst80]. The special theory of semijoins - a tool for the execution of distributed query transactions - was presented in [Bern81a], while the separation of query transformation and optimization - a tool for the analysis and synthesis of distributed query approaches - was presented in [Chu82]. A general theory of query optimization may be found in [Ceri83b].

- Optimizing updater transactions is not a possibility. There is no choice - from data fragmentation, data replication or data location - with regard to where to write new data-item values.
- **Data Fragmentation, Data Replication and Data Location:**

  - Optimizing *item allocation* is both a possibility and a necessity. There is a choice with regard to where to store *user-data* - representing the database state - and *system-data* - representing for example the database directory, and the speed effects on the global access of data are considerable. This is a design topic; i.e. it concerns efficient *data distribution*.

  A first study of data fragmentation introducing the so-called views came with [Daya78], while later works refined the ideas respectively introducing snapshots ([Adib80]), derived relations ([Adib81]) and multi relations ([Ceri83b]). Two proposals concerning how to make fragments of user-data were presented in [Chan80] and [Ceri83a]. One specific proposal concerning where to place such fragments may be found in [Chen80], while a generic comparison of several such proposals was given in [Dowd82]. A comparison of several proposals about how to make and where to place fragments of system-data also came with [Chu76].

- **Global Data Transparency:**

  - Hiding *item distribution* is highly recommended. It deals with how to actually access data. The users of a distributed database should not have to bother about which fragments each data-item is split into, how many replica each such fragment is copied in, or the *locating* site of each such replica.

- **Local Site Autonomy:**

  - Distributing *process control* is nearly an absolute requirement. It deals with how to actually reach consensus with regard to accessing data. The owners of the separate parts of a distributed database should be able to control the use of the resources at their corresponding sites. Having independent decision makers again leads to the requirement of the *atomic commitment*. (Refer to the implementation of recoverability for atomic transactions above). Here it means that before an operation relating to several independent sites may be carried out, all the involved sites have to be inquired with regard to their *willingness* to perform the corresponding actions. All of them still have to respond positively before an actual decision to go through with the operation may be taken. (Again see Section 3.2.2).

Let us now only introduce a simple and *common mechanism*; i.e. the **2PhaseCommitment (2PC)** - an atomic commitment protocol from [Gray78]:
2PC: No Decision may be Taken before
All Decision Makers have been Inquired

1. Phase = Agreeing — Commit-Point

2. Phase = Acting

The rule says that the sites involved in a transaction have
to be consulted so that at a specific point in time (the
commit-point), a decision may be taken that all sites are
able and willing to live with. By the way, compare this
mechanism with the textually similar 2PhaseLocking (2PL)
mechanism of Section 2.2.7.

- Common Interface:

- Offering a unified data model and a unified query language to
  the users of a distributed database is highly recommended.
  For a homogeneous system this is an easy task per definition,
  while for a heterogeneous system this is in practice a
tremendous task. Where applicable, the relational data model
and a relational query language are usually chosen.

0 Commission Errors:

- Achieving the so-called byzantine agreement among separate
  sites of a distributed system is not arrangeable. This
  concerns deliberate faults of malicious parts - in contrast
  with non-deliberate faults of non-malicious parts (or
  omission errors). The area was introduced in [Peas80], and a
  survey of approaches was given in [Str083].
3.1.5 System Structure

From the analysis in the previous section it is possible to outline an implementation of a distributed database system as a hierarchy of levels, as given in Fig. 3.2.

![Diagram](image)

**Fig. 3.2. Software Architecture of a Distributed Database System.**

The indicated layers deserve some extra explanatory comments.

- **Item Binding:**

  This level includes a translation from a locally unique data specification to a globally unique data specification. Thus one decides what logical data-item is indicated - through information about user-data fragmentation in the directory.

- **Data Space Determination:**

  This level includes a translation from a logically valid data specification to a physically valid data specification. Thus one decides where stored data-item copies corresponding to an indicated data-item, are placed - through information about user-data replication in the directory.

- **Data Manipulation Efficiency - Data Distribution Efficiency:**

  This level dynamically employs information with regard to user-data location from the directory to decide which stored data-item copy corresponding to an indicated data-item, should be accessed. Through its continuous employment of
user-data information, this level also dynamically generates information with regard to user- and system-data fragmentation, replication and location to the directory and directory-directory. Thus it carries out a continuous reorganisation of user- and system-data; i.e. adapting to the actual access of data.

- Consistency Preservation & Reliability Assurance - Consensus Achievement:

  This level abstracts from parallel use and erroneous behaviour of the system database, and from local decision rights in the separate sites - basically to be dealt with in the rest of this chapter.

- System Perception Consistency:

  This level abstracts from the non-coincidence of event productions with event observations - also to be commented upon in the rest of this chapter.
3.1.6 **System Details and Basics**

First, in Fig. 3.3 we will expand on the model of a distributed database system from Fig. 3.1 in Section 3.1.2 a little further.

![Diagram of a distributed database system](image)

**Fig. 3.3. Low-Level Model of a Distributed Database System.**

The **processing media** correspond to cpus, the **buffer medium** to a cache (primary memory) and the **storage media** to disks (secondary memory). Further the **input/output media** correspond to terminals and the **links** to communication channels.

We have already indicated that there are usually several terminals associated to each single site in a centralized or distributed database system.

It is important to emphasize that in each single site, it is also normal to implement the processing capacity with **multiple cpus** and the storage capacity with **multiple disks**.
This duplication of processing power and storage power not only between the multiple sites but also within each single site has some essential implications:

- **True parallel activities** may occur even within a single site because of the multiple processing media. But this happens with **instant signal observability** as the local memory and clock are still shared. This is contrary to between sites where we have true parallel activities without instant signal observability - stemming from the memory and clock not being shared globally.

- **Partially erroneous parts** may occur even within a single site because of the multiple storage media. But this happens with **dependent decision makers** (non-autonomous units) as the local database system does have unique control. This is contrary to between sites where we have partially erroneous parts with independent decision makers (autonomous units) - stemming from the database system not having unique control globally.

The more **indirect effects** of these facts again are important:

- **2PhaseLocking (2PL)** (or a similar mechanism) is needed both per local site and per **global system** as all processing, input/output and communication media - i.e. the global database accessing units, may work at the same time.

- **2PhaseCommitment (2PC)** (or a similar mechanism) is needed both per global system and per **local site** as any processing, buffer or storage medium - i.e. the local database storage units, may crash at any time.

So even though the specific problems and issues are different for a single site versus multiple sites, some **generic areas** need **similar attention and treatment**.

Second, in Fig. 3.4 we will look at a distributed database system from a slightly **different angle** than in Fig. 3.1.

The **circles** represent processing units, while the **squares** represent storage units.

Initially, we will emphasize that the sites are usually grouped somehow. On each out of two levels there is a set of mutually independent sites or cohorts. However the sites on the bottom level are divided into subsets so that the sites of each subset depend on a specific site on the top level. Thus each independent cohort on the upper level uniquely controls its corresponding dependent cohorts on the lower level, while there is no site uniquely controlling the independent cohorts. A **hierarchy of sites** of this type may of course be generalized further.
Further, it is necessary to explain the different types of arrows used:

- A quadruple arrow represents a transaction initiation from an external user to an independent cohort.

- A triple arrow represents a subtransaction initiation between independent cohorts. The initiating site might send a query request and await the receipt of the results, or send an updaters request with parameters and await the receipt of an acknowledgement. The query or updaters request could either include a program-name or contain the program-code.

- A double arrow indicates an action command from an independent to a dependent cohort. The commanding site might ship an item-retrieval awaiting the result in return, or ship an item-update with a value awaiting an acknowledgement in return.

- A single arrow indicates a subaction command from a processing unit to a storage unit. The commanding unit requires the reading of a value or writing of an item.
Finally, let us investigate the procedure for transaction termination; i.e. we assume an incoming commit-request from the user:

1) The independent cohort supervising the transaction must see to that it and all the other independent (updater) cohorts involved agree on a unique decision with regard to their willingness either to commit or abort the transaction. Both [Date83] and [Ullm82] naturally claim that this many-control-many situation requires a 2PC mechanism. This corresponds to a site autonomy implementation within the full line encircling in the figure.

2) Point 1) above implies that each independent cohort must see to that it and all its dependent (update) cohorts involved reach a common result with regard to their ability either to commit or abort the transaction. [Date83] again naturally claims that this one-controls-many situation requires a 2PC mechanism. This corresponds to a global recovery implementation within the dashed line encircling in the figure.

3) Point 2) above again implies that each processing unit must make its one/many storage unit(s) able to carry out all or none of the corresponding writes. [Ullm82] also claims that this one-governs-one/many situation requires a 2PC similar mechanism. Effectively this is acceptable - especially in view of the discussion of partially erroneous parts earlier in this section, but the terminology may be slightly confusing. This corresponds to a local recovery implementation within the dotted line encircling in the figure.

We rather prefer to designate the mechanism needed in this case 2PhaseWriting (2PW):

\begin{verbatim}
2PW: No Item may be Written to the Database before
All Items have been Written to the Log

1. Phase = Log-Writing — Secure-Point
2. Phase = Database-Writing
\end{verbatim}

The rule says that the items written by a transaction have to be accessed so that at a specific point in time (the secure-point), copies of all new values are in the log, while all old values are in the database. Compare this with the 2PC definition in Section 3.1.4 (and the 2PL definition in Section 2.2.7). Also compare this with the undo-rule and the redo-rule of the LogWriteAhead protocol in Section 2.4.2. The undo-rule is per action, while the 2PW-rule is per transaction.
We may thus conclude that:

- 2PW is a recovery mechanism for a single site
- 2PC is a recovery mechanism and a site autonomy mechanism for multiple sites
- 2PL is a concurrency control mechanism for a single site and for multiple sites
3.1.7 Correctness with regard to Failures and Interference

We will sum up by successively giving correctness criteria for gradually more complex cases.

First, let us state correctness criteria for a single, isolated and terminating transaction. Thus we discard failure, site autonomy and interference aspects. (Case 1).

a) Single Site Only:
   - Read the items to be retrieved at that site
   - Write the items to be updated at that site

b) Multiple Sites with Fragmentation but with No Replication:
   - Read the items to be retrieved at the corresponding sites
   - Write the items to be updated at the corresponding sites

c) Multiple Sites with Fragmentation and with Replication:
   - Read one copy of each item to be retrieved at the necessary sites
   - Write all copies of each item to be updated at the corresponding sites

The writes of a transaction in subcases b) and c) may either be carried out immediately or deferred some time because of efficiency reasons. Deferring a write in subcase b) requires the use of an extra buffer as temporary storage, while in subcase c) one of the copies themselves may play this role. Using one copy like this, opens the way for two options for the other copies. Either they will always be updated - though not immediately like the main copy - resulting in a primary copy plus secondary copies configuration, or they will only sometimes be updated - periodically refreshing them from the main copy - resulting in a master item plus slave snapshots configuration (see [Adib80] and [Adib81]). This last variant actually removes some of the replication aspects per se. (Alternatively it may lead to some breaks with standard serializability).

In the most general situation - i.e. subcase c), we may thus at least distinguish between three options:

   - Write all the copies immediately
   - Write the primary copy first and write the secondary copies later
   - Write the master item always and write the slave snapshots sometimes
For a corresponding read the option is:

- Read any copy
  
  (not necessarily the same as an eventual
  primary copy or master item)

Observe also the following:

- **Reading** several items requires reading all of them, while
  reading several copies of an item requires reading only one
  of them.

- **Writing** several items generally enforces any consistency
  among items, while writing several copies of an item
  specifically enforces equality consistency within an item.

Second, let us state correctness criteria for a single, isolated
transaction. Thus we emphasize failure and site autonomy aspects, but
discard interference aspects. (Case 2).

a) Single Site Only:

Having only **one decision maker** and risking only total
failures of **items** require the addition of a local recovery
mechanism like 2PW only. Any transaction retrieving/updating
items at the non-functioning site must halt.

b) Multiple Sites with Fragmentation but with No Replication:

Having **several independent decision makers** and risking
partial failures of **items** require the addition of (2PW +) a
global recovery and site autonomy mechanism like 2PC. Any
transaction only retrieving/updating items at sites still
functioning may continue.

c) Multiple Sites with Fragmentation and with Replication:

Having **several independent decision makers** and risking
partial failures of even **copies of items** require the
assurance of (2PW + 2PC +) a common failure/non-failure view
of item copies. Any transaction only retrieving/updating
items having at least one copy each at sites still
functioning may continue.

The **messages** exchanged between normal sites in subcases b) and c) -
i.e. containing item-values/item-copy-values or control-information,
now have to overcome the possibility of the sender and receiver being
up/down at different times. In both subcases one alternative is to
employ extra **spooling-sites**, another to use general **polling-actions**.
In subcase c) a third alternative is to introduce specific copier-transactions (for item-copy-values).

Further the different update-options in the previous subcase lc) now have to be adapted to the chances of failures. Requiring that absolutely all copies of an item should be updated before a transaction may continue, would decrease the availability of the system instead of increasing it. Thus we must settle by updating all available copies. Requiring that a specific primary copy or master item should be updated before the other copies/snapshots, would likewise be counter-effective. Thus we must allow the designation of a primary copy/master item to change over time. We end up with:

- Write all the available copies immediately
- Write the current primary copy first and write the secondary copies later
- Write the current master item always and write the slave snapshots sometimes

And for a corresponding read the option is:

- Read any available copy
  (not necessarily the same as an eventual current primary copy or current master item)

(The common failure/non-failure view of item copies also needed, will be further dealt with in Section 3.3.2).

With reference to Fig. 2.9 in Section 2.2.3 note also the following:

- Each recovery manager separately carries out the local transaction management with regard to failures (e.g. 2PW).
- All transaction managers collectively carry out the global transaction management with regard to failures and site autonomy (e.g. 2PC). One site plays the role of coordinating cohort, and the others the roles of participating cohorts.

Third, let us state correctness criteria for multiple transactions. Thus we emphasize failure, site autonomy and interference aspects. (Case 3).

a) Single Site Only:

All transaction conflicts being detectable each on a specific item at that site requires the addition of (2PW+) a concurrency control mechanism like 2PL only.
b) Multiple Sites with Fragmentation but with No Replication:

Different transaction conflicts being detectable each on a specific item but at separate sites requires no addition (besides 2PW + 2PC + 2PL) - from correctness considerations.

c) Multiple Sites with Fragmentation and with Replication:

Some transaction conflicts being non-detectable each on a specific item copy among the separate sites requires the assurance of (2PW + 2PC + common failure/non-failure view of item copies + 2PL +) a common control-copy or fixed control-point for item copies.

The different update-options in the previous subcase 2c) now have to be further adapted to the chances of interference. For each transaction conflict to manifest itself at at least a common control-copy, we may require that an updating transaction X-locks all copies of the item to be written, while a retrieving transaction S-locks any copy of the item to be read - or that both an updating and a retrieving transaction X- or S-lock a predetermined copy of the item to be accessed. For each transaction conflict to manifest itself at a fixed control-point, we may require that any transaction for each item to be accessed locks a specific data-variable associated with that data-item. This set of data-variables may be centralized, distributed or located in any hybrid way - usually totally separated from their corresponding item copies. The availability and currency issues apply here too, so we end up with:

- X-lock all the available copies or S-lock any available copy
- X-lock or S-lock one current common copy
- X-lock or S-lock one current fixed point

(Some other options will be further dealt with in Section 3.3.1).

For a write-action itself the options still become:

- Write all the available copies immediately
- Write the current primary copy first and write the secondary copies later
  (the current primary copy is not necessarily the same as an eventual X-locked current common copy)
- Write the current master item always and write the slave snapshots sometimes
  (the current master item is not necessarily the same as an eventual X-locked current common copy)
And for a read-action itself the option still becomes:

- Read any available copy

(not necessarily the same as an eventual S-locked copy and not necessarily the same as an eventual current primary copy or current master item)

Again with reference to Fig. 2.9 note also the following:

- Each and all concurrency managers separately and collectively carry out the local and global transaction management with regard to interference (e.g. 2PL). The global aspects (e.g. putting together conflict information for deadlock detection) may be centralized, distributed or performed in any hybrid fashion.
3.2 Fragmented but Non-Replicated Systems

First, we will look at ways to implement serializability and recoverability for multiple sites with fragmentation but with no replication. We are thus combining material from Sections 3.1.7, 2.3.2, and 2.4.2.

3.2.1 Concurrency Control

Initially, reads and writes are only local operations. This implies that (non-replicated) multiple-site concurrency control may be implemented basically as single-site concurrency control. All the previous mechanisms to achieve one-after-the-other atomicity from Section 2.3.2 thus have their distributed counter-parts. Hence there are Distributed-2PL, Distributed-T0, Distributed-SGT and Distributed-C.

An extra possibility is the option to dissect concurrency control on different geographic locations. (See also Section 2.3.2 concerning a corresponding dissection on different types of conflicting operations):

One may separate concurrency control at distinct sites from each other. Hence it is possible to synchronize accesses at one site separately and differently from accesses at another site - provided an arrangement is added to make all the resulting serialization orders consistent. Any of the four basic mechanisms may be used at each of the different sites - giving any number of mixed cases.

Further, we will discuss some of the consequences of going from a centralized to a (non-replicated) distributed setting on the four basic mechanisms:

a) Distributed-2PL (D-2PL):

Locking of items in a distributed system is still a local problem, while deadlocks occurring in a distributed system naturally is a global problem.

Such global occurrences may basically be solved with the same three approaches as introduced in Section 2.3.2: i.e. deadlock avoidance, prevention and detection.

In a distributed setting there is a clear difference between avoidance/prevention techniques - giving less concurrency for less overhead, and detection techniques - giving more concurrency for more overhead. While in a centralized system one may often afford the overhead connected with choosing deadlock detection, in a distributed system one may often prefer deadlock avoidance/prevention to limit the overhead.

Detecting deadlocks in a distributed setting may be carried out in several different ways:
i) Centralized in one site.
   For an example, see [Gray78].

ii) Hierarchically within sets of sites.
    For an example, see [Mena79].

iii) Distributed among all the sites.
    For two examples, see [Ober82] (with optimized information exchange) and [Chan83] (without information exchange).

Detecting deadlocks in a distributed system will also be complicated through the problem of the so-called phantom deadlocks associated with the need to know not only who blocks whom, but also when the blocking occurs. Thus there is a real-time constraint that does not exist in a centralized system.

b) Distributed-TO (D-TO):

Ordering of item accesses in a distributed system is still a local problem, while timestamps valid for a distributed system naturally is a global problem.

Such global timing may be solved by the logical clocks concept of [Lamp78] mentioned in Section 3.1.4. Hence to the local physical time - giving a partial ordering of events, one concatenates a global site identification - giving a total ordering of events.

c) Distributed-SGT (D-SGT):

Testing of the partial serialization graphs in a distributed system is still a local problem, while arranging for a total serialization order is a global problem.

Such global ordering requires extra inter-site communication, and the unification process is subject to an absolute real-time constraint.

d) Distributed-C (D-C):

Certification for each separate site inherits the local problems of the basic mechanism - specifically depending on whether we have a D-2PL-, D-TO- or D-SGT-Certifier, while certification for all sites together adds the global problem of all-or-none unity.

Such global unification requires an atomic commitment type protocol like 2PC.

The choice of a concurrency control mechanism may (as the choice of a deadlock resolution technique) heavily depend on whether considering a
centralized or distributed setting. There is for example again a clear
difference between D-2PL - giving more concurrency for more overhead,
and D-TO - giving less concurrency for less overhead. Once more, while
in a centralized system one may often afford the overhead connected
with choosing D-2PL, in a distributed system one may often prefer D-TO
to limit the overhead. However such an opinion was more prevailing
earlier. A discussion of distributed concurrency control complexity
may be found in [Kane85].

As abortions are very costly in a distributed setting, there has also
been a tendency to choose the conservative more than the aggressive
variants of the concurrency control mechanisms for distributed
systems. (Again see Section 2.3.2).

Finally, in Sections 2.3.2 and 2.4.1 we treated and commented upon a
specific interconnection between local concurrency control and local
recovery. That concerned preparing for limited effects of possible
abortions versus dealing with actual consequences of occurring
abortions. There is also a certain coupling between global concurrency
control and global recovery even for the non-replicated case. From the
way we present the material - i.e. covering only interference topics
in this section and covering added failure and site autonomy topics in
the next section, we leave the treatment of the associated rule to
that section.
3.2.2 RECOVERY

Initially, commits and aborts (plus restarts and resets) are both local and global operations. This implies that (non-replicated) multiple-site recovery may partly be implemented basically as single-site recovery. All the previous mechanisms to achieve all-or-nothing atomicity with regard to local total failures and one decision maker from Section 2.4.2 thus have their distributed counter-parts. Hence there are global abortions, global restarts and global resets possibly with global checkpoints and global dump checkpoints. This concerns dealing with global partial failures that do occur.

One extra necessity is all-or-none atomicity with regard to global partial failures. This leads to the need for atomic commitment protocols like 2PC. It concerns preparing for global partial failures that might occur.

Another extra necessity is all-or-none unity with regard to several independent decision makers. This also leads to the need for atomic commitment protocols like 2PC.

Further, we will discuss some of the consequences of going from a centralized to a (non-replicated) distributed setting on the three basic mechanisms and their optimizing techniques:

- Transaction-Failures:

  Error conditions relating to a specific transaction require a global abortion of this transaction. This implies a local abortion of the transaction both on the site where the error condition is detected/enforced, and on all the other involved sites.

- System-Failures:

  Loss of volatile storage at one site in the system requires a global restart of all sites in the system. This implies a local restart at the failed site plus a global abortion of each transaction that thus had to be undone at the specific site (with consequences at all the other involved sites).

- Media-Failures:

  Loss of non-volatile storage at one site in the system requires a global reset of all sites in the system. In both the good and bad cases this implies a local reset at the failed site plus a global abortion of each transaction that thus could not be redone at the specific site (with consequences at all the other involved sites). As before, for the good case this will only include previously non-committed transactions, while for the bad case this may include both previously non-committed and previously committed transactions. Global dump checkpoints (i.e. corresponding to consistent sets of committed transactions) would considerably ease especially the reconstruction work in the bad case.
Efficiency Considerations:

Global checkpoints and global dump checkpoints may be implemented either as an extension of the 2PhaseCommitment (or any other atomic commitment protocol) - see [Duda80] for such a small-grain example, or as an addition of a 2PhaseCommitment similar technique - see [Kuss82] for such a coarse-grain example.

As an interlude, we also have to investigate the effects of failures of single units on a system of multiple units.

One classification of failures possibly occurring in a (non-replicated) distributed system is the following:

- Site Failures:

  This concerns failures happening in the processing and storage nodes of the network. Each site is considered to be either up or down. Hence it is either functioning perfectly or not functioning at all. A failing site halts without further destroying any of its resources. For N nodes we have:

  1 Site Down = Single (Partial) Failure

  2 N-1 Sites Down = (Multiple) Partial Failure

  N Sites down = (Multiple) Total Failure

- Link Failures:

  This concerns failures happening in the transmitting edges of the network. Each link is considered to be either up or down. Hence it is either functioning perfectly or not functioning at all. A link up will never loose, garble, duplicate or missequence transmitted messages.

From this division another classification of failures may be derived which is more appropriate here:

- Site Failures:

  1 N Sites are Down.

  This is as above.

- Communication Failures:

  2 (N) Sites are Unable to Exchange Messages.

  This may be due to link failure(s) and/or other site failure(s). A message that is undeliverable, is supposed to
be dropped and not buffered in the system. A communication failure is thus supposed to be detected by a timeout on an unreceived acknowledgment concerning a message that is sent.

Following this approach, the very important aspect of network partitioning may occur in two different ways:

- Physical Network Partitioning:

  A true communication failure will imply that any/all communication path(s) between 2 or more sites up must have broken down - in general as a combination of site and link failures.

  Effectively the set of all sites up have been partitioned into 2 or more separate sets of sites with any specific site in any specific set being unable to communicate with any other site in any other set. Hence the distributed system has been divided into isolated partitions.

- Logical Network Partitioning:

  A false communication failure may occur if 1 or more sites up use too small timeout periods to detect such failures.

  Thus the monitoring and broadcasting of site-state and partition-state changes from [Walt82] mentioned in Section 3.1.4, ought to be used instead of a timeout detection.

Both these types of network partitioning are manifested in our model as communication failures, but we will only concentrate on the physical one.

A site failure will make the items stored at that site unavailable to any other site. A communication failure will also make the items stored at a site up in a partition unavailable to any site in any other partition.

For a specific transaction originating at a certain site in a non-replicated system, all items to be accessed by it must be available from that site for the transaction to be able to execute and terminate normally. However any write may be buffered at the originating site.

Next, we will discuss some of the aspects concerning atomic commitment protocols.

The main point is that one coordinating and several participating cohorts must both reach a common result with regard to their ability either to commit or abort a specific transaction - i.e. this concerns failures, and agree on a unique decision with regard to their willingness either to commit or abort the specific transaction - i.e. this concerns site autonomy.
This implies that all corresponding transaction managers at the involved sites have to vote over the final outcome of a certain transaction.

A participating transaction manager must vote no to normal termination if it is not able to commit the transaction - e.g. stemming from a site failure (corresponding to a system- or media-failure) which has occurred earlier, or if it is not willing to commit the transaction - e.g. stemming from any motive related to local site autonomy. In all other cases the participating transaction manager might vote yes to normal termination.

The coordinating transaction manager has to organize this voting process. Hence it requests votes, collects votes and makes a decision from the votes. Of course it has to vote too, either yes or no. All the votes have to be yes for a commit to be the final decision. A site or communication failure which currently occurs, may even cause a vote not to be deliverable. In such a case the coordinating transaction manager must make the final decision an abort.

Once more with reference to Fig. 2.9 in Section 2.2.3, a final commit decision thus requires the following (anticipating N involved sites):

1) The coordinating transaction manager must have voted yes, and N-1 yes votes must have been sent and received from the participating transaction managers.

Besides its own ability and willingness issues, for a specific transaction manager to be capable of voting yes,

2a) the corresponding concurrency control manager has to produce at least recoverable schedules (see Section 2.2.8), and

2b) the corresponding recovery manager has to obey at least the redo-rule (see Section 2.4.2).

As examples of atomic commitment protocols we will present:

a) 2PhaseCommit (2PC):

In its basic form - i.e. anticipating no failures during the commitment processing, we have:

**Coordinator:**

**Participants:**

1. **Phase:** Send for Votes
   
   Receive the Votes
   
   Decide from Votes

2. **Phase:** Command the Result

   Execute Commit/Abort
This is the mechanism already introduced in Section 3.1.4. For a full treatment of it which also covers failures occurring during the commitment processing, see [Gray87]. Two optimized versions of 2PC are described in [Mohr83].

b) 3PhaseCommit (3PC):

Again in its basic form - i.e. anticipating no failures during the commitment processing, we have:

Coordinator: Participants:

1. Phase: Send for Votes Respond Yes/No
   Receive the Votes
   ----- Decide from Votes -----

2. Phase: Prepare for Result Notice Result
   Receive the Acks

3. Phase: Command the Result Execute Commit/Abort

In this mechanism an extra phase is introduced to assure that no cohort executes a commit while any other cohort(s) may be uncertain about the final decision (commit/abort). This is done to minimize blocking; i.e. to reduce the periods where an uncertain cohort cannot proceed until a specific communication failure between it and some other cohort(s) has been repaired. For a full treatment of the mechanism which also covers failures occurring during the commitment processing, see [Skee81a]. An optimized version of 3PC is described in [Skee82].

We will also mention a third atomic commitment protocol: i.e. the 4PhaseCommit (4PC) from [Hammm80]. In this mechanism two extra phases are introduced to establish and use backup-sites for the coordinating cohort. With regard to the amount of blocking, we may roughly say that:

2PC > 4PC > 3PC

A detailed analysis of several atomic commitment protocols with regard to blocking is given in [Coop82], while an in depth general discussion of atomic commitment protocols with regard to what is theoretically achievable or not may be found in [Skee81b].
Finally, let us show how global concurrency control and global recovery are coupled (even for the non-replicated case). We will use D-2PL as an example:

- **Locks** at any site must be acquired before the first (1.) phase of atomic commitment.

  Deadlocks may for example occur, and this may influence the voting result.

- **Locks** at a site whose corresponding transaction manager has voted yes, may not be released until after the last (2./3.) phase of atomic commitment.

  The final result is unknown; i.e. either commit or abort.

  This is an extension of the need to produce at least recoverable schedules for a transaction manager to be capable of voting yes itself, see above.

- **Locks** at a site whose corresponding transaction manager has voted no, may be released before the last (2./3.) phase of atomic commitment.

  The final result is known; i.e. abort.

LockUntilEnd (see Section 2.3.2) is thus again a natural choice instead of 2PhaseLocking.
3.3 Fragmented and Replicated Systems

Second, we will look at ways to implement serializability and recoverability for multiple sites with fragmentation and replication. We are thus again combining material from Sections 3.1.7, 2.3.2 and 2.4.2.

3.3.1 Concurrency Control

Stemming from one logical item being represented as several physical item copies, we need a common physical control-copy or fixed separate control-point for the item copies. Thus item-conflicts are manifested as copy-conflicts or point-conflicts.

Initially, added to the non-replication mechanisms from Section 3.2.1, we have to make arrangements for such common control-copies or fixed control-points. We will again use D-2PL as an example. The list of variants from Section 3.1.7 may for example be extended into:

i) X-Lock All Copies (C)
   S-Lock Any Copy (I)

ii) X-Lock Copies corresponding to a Majority/Quorum
    S-Lock Copies corresponding to a Majority/Quorum

    This is a variant of i).

    Assigning a general weight \( W_i \) to each copy,
    a quorum \( Q = (\sum_{i=1}^{C} W_i)/2 + 1 \), and
    assigning the unity weight 1 to each copy,
    a majority \( M = C/2 + 1 \).

iii) X-Lock Copies corresponding to K Weight Units
    S-Lock Copies corresponding to S-K+1 Weight Units

    This is a generalization of i) and ii).

    For general weights we have
    \( S = \sum_{i=1}^{C} W_i \) and \( S \geq K \geq Q \), and
    for unity weights we have
    \( S = C \) and \( C \geq K \geq M \).

iii) corresponds to

minimum 1 and maximum \( C/2 + 1 \)
non-specific common control-copies for reads vs. writes, and
minimum 1 and maximum C
non-specific common control-copies for writes vs. writes.

iv) X- and S-Lock the Primary Copy

v) X- and S-Lock the Master Item
   iv) and v) correspond to a specific common control-copy.

vi) X- and S-Lock at a Central Site
   vi) corresponds to a specific fixed control-point.

Locking of several copies may (as locking of several items) lead to
deadlocks. One simple approach for this problem is to adapt one of the
deadlock avoidance techniques from Section 2.3.2; i.e. use
linearization to order copies and force transactions to do locking
according to this.

Finally, this possibility to keep several identical copies of each
item in space is a very important aspect. In [Bern86] and [Bern87a] a
general replicated-data serializability theory was developed. The
theory is based on a concept introduced in [Atta84], and this concept is fairly recoverability oriented. This is only natural as the theory has to cope with copies of items being created and failing again possibly several different times. So the theory for the replicated case must deal with an extra coupling existing between global concurrency control and global recovery. This is in addition to the one for the non-replicated case mentioned in Section 3.2.1 and treated in Section 3.2.2. From the way we present the material - i.e. covering only interference topics in this section and covering added failure and site autonomy topics in the next section, we leave the treatment of the associated notions and algorithms to that section.
3.3.2 Recovery

Stemming from one logical item being represented as several physical item copies, we need a common failure/non-failure view of the physical item copies corresponding to a logical item. Thus the perception of creations and failures of item copies representing a unique item, is the same across sites.

Initially, the basic methods from the previous section (using D-2PL for concurrency control) must be adapted to the availability and currency consequences like in Section 3.1.7:

i) X-Lock All Available Copies ($C_a$)
   S-Lock Any Available Copy  (1)

ii) X-Lock Available Copies corresponding to a Majority/Quorum
    S-Lock Available Copies corresponding to a Majority/Quorum
    Quorum $Q = (\sum_{i=1}^{C_a} W_i)/2 + 1$, and
    majority $M = C_a/2 + 1$.

iii) X-Lock Available Copies corresponding to K Weight Units
     S-Lock Available Copies corresponding to S-K+1 Weight Units
     $S = \sum_{i=1}^{C_a} W_i$ and $S \geq K \geq Q$, or
     $S = C_a$ and $C_a \geq K \geq M$.

iv) X- and S-Lock the Current Primary Copy

v) X- and S-Lock the Current Master Item

vi) X- and S-Lock at a Current Central Site

As an interlude, we have to investigate the effects of site and communication failures in a replicated distributed system too.

A site failure will make the copies of items stored at that site unavailable to any other site. A communication failure will also make the copies of items stored at a site up in a partition unavailable to any site in any other partition.

For a specific transaction originating at a certain site in a replicated system, at least one copy of all items to be accessed by it must be available from that site for the transaction to be able to execute and terminate normally. However any write may be buffered at the originating site.
Finally, added to the non-replication mechanisms from Section 3.2.2, we have to make arrangements for the common failure/non-failure view of the item copies.

In this context we will distinguish between a case where only site failures are supposed to occur, and the case where both site and communication failures are allowed to occur.

- Site Failures (= No-Partitioning):

  The sites up will constitute a single united partition. Basically we may allow all sites up (in the one partition) to do updating without any danger of corrupting the database consistency.

- Site + Communication Failures (= Partitioning):

  The sites up will possibly constitute multiple isolated partitions. Basically we must allow maximum all sites up in one partition (out of the many partitions) to do updating to avoid the danger of corrupting the database consistency.

In this context we will also distinguish between a case where failures are supposed to occur only between transaction processing, and the case where failures are allowed to occur both between and within transaction processing.

- Failures between Transaction Processing (Static Conditions):

  No changes with regard to availability/non-availability of sites (i.e. copies of items) will occur as transactions are executing. A transaction may check site- and partition-states once at transaction initiation.

- Failures between and within Transaction Processing (Dynamic Conditions):

  Some changes with regard to availability/non-availability of sites (i.e. copies of items) will possibly occur as transactions are executing. A transaction must test site- and partition-states continuously during transaction processing.

We end up with four different groups. Where it is necessary in the discussions of the coming examples, we will once more use D-2PL for concurrency control:

- Static No-Partitioning case:

  In this hypothetical case all the six variants above [i)-vi)] are applicable.
- Dynamic No-Partitioning case:

For this case failures and recreations of item copies (corresponding to crashes and repairs of sites) have to be detected and acted upon. With regard to the six variants above, the detected site-state changes must for each involved item in i) to iii) lead to a recounting of $C_a$ and a recalculation of $Q/M$ and $S$ - concerning the copies now available from the acting site in the one partition. Likewise, they must for each involved item in iv) to vi) lead to a reevaluation of the current distinguished copy/item/site - among all sites now up in the one partition.

An i)-type example is the available copies (AC) algorithm of [Bern84]. After an access-phase of the i)-type, a validation-phase succeeds. If copies previously unavailable at attempted writing then are available, or copies previously available at reading or writing then are unavailable, the transaction has to be aborted - to avoid non-serializability.

- Static Partitioning case:

Compared to the static no-partitioning case, in this hypothetical case a transaction must initially find out whether its originating site belongs to the one partition to be allowed to do updating or not. Rather we make it initially check that a majority or quorum of copies is available from the originating site for each item to be accessed. This majority or quorum of available copies in that specific partition should be relative to the total number of available copies in all existing partitions. But as this number is non-decidable, we have to settle by checking relative to the total number of copies in the system (currently available and unavailable). With regard to all the six variants above, first a transaction establishes a total majority or quorum among all partitions for each item to be accessed. Then in i) to iii) each $C_a$ refers to copies within the specific partition, and $M$ or $Q$ thus refers to a partial majority or quorum within that partition. Likewise, in iv) to vi) each current distinguished copy/item/site refers to one within the specific partition.

- Dynamic Partitioning case:

Compared to the dynamic no-partitioning case, for this case a transaction must "continuously" make sure that its originating site belongs to the one partition to be allowed to do updating. Again rather we make it "continuously" test whether a majority or quorum of copies is available or not from the originating site for each item to be accessed. This majority or quorum of available copies in that specific partition should once more be relative to the total number of available copies in all existing partitions. But as this number is non-decidable, we have to settle by testing relative to the total number of copies in the system (currently available and unavailable). With regard to all the six variants above, effectively a transaction controls a total majority or quorum among all partitions for each item.
to be accessed. Further in i) to iii) each $C_a$ refers to copies within the specific partition, and $M$ or $Q$ thus refers to a partial majority or quorum within that partition. Likewise, in iv) to vi) each current distinguished copy/item/site refers to one within the specific partition.

Some i)- + ii)-type examples are the 

**quorum consensus (QC) algorithm of [Giff79], the missing writes (MW) algorithm of [Eage83] and the virtual partition (VP) algorithm of [ElAb85].** In QC the controlling of a total quorum and the locking of a partial quorum are combined by using the number corresponding to the total quorum for both purposes. Actually both retrieving and updating items also require accessing quorums of copies. The single access-phase of type ii) employs version-numbers on item copies to avoid non-serializability. Specifically, retrieving an item requires reading a quorum of copies, while updating an item requires first reading then writing a quorum of copies. Each write will set a version-number which is higher than the highest of the version-numbers from all reads. To decrease the inefficiency of QC, MW combines the possibilities of AC and the necessities of QC. It functions as AC (i)-type) during failure-free periods, while it functions as QC (ii)-type) during transition-periods between failure-free periods. A diversion from a failure-free period is detected through attempted writes or attempted validation of writes in AC. To further increase the efficiency of MW, VP actually does not access quorums of copies but only assures the existence of quorums of copies (thus forming views reflecting available copies). Specifically, before an access-phase of type i) (resembling the access-phase of AC), an assurance-"phase" of type ii) precedes (resembling the access-part of QC). Any discrepancy between view-perceptions between the two phases will lead to the abortion of the transaction and the formation of a new view.

Descriptions of other examples of algorithms fitting the divisions of our model may be found in [Bern80c] (i)-type), [Thor79] (ii)-type), [Ston79] (iv)-type), and [Als87] plus [Mena80] (vi)-types).

Yet other examples of algorithms are described in [Elli77] (ring variant) and [Mino82] (token variant).

[Skee84] presents a method for the analysis of updating done in several partitions with regard to when such updating cannot lead to non-serializability - and thus might be allowed to occur. With regard to even allowing updating in several partitions that can lead to non-serializability, [Park83] indicates how to detect that such inconsistencies among partitions have occurred. [Garc83] indicates how to resolve such inconsistencies among partitions - after they have occurred and have been detected. [Davi85] surveys different approaches used in replicated distributed systems, both with regard to allowing single partition updating only and with regard to allowing multiple partition updating even.
3.4 System Examples

Let us end this chapter with relevant information about specific examples of real distributed database systems. Table 3.1 covers several examples of homogeneous systems employing a long-haul network, while Table 3.2 covers single examples of respectively heterogeneous systems employing a long-haul network and homogeneous systems employing a local-area network. These tabulations have been inspired by [Ceri84].

Table 3.1. A comparison of some Long-Haul Network and Homogeneous Systems.

<table>
<thead>
<tr>
<th>Topic</th>
<th>SDD-1</th>
<th>R*</th>
<th>Distributed Ingress</th>
<th>Porel</th>
<th>Sirius-Delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Model</td>
<td>Relational</td>
<td>Relational</td>
<td>Relational</td>
<td></td>
<td>Relational</td>
</tr>
<tr>
<td>Data Fragmentation Allowance Type Support</td>
<td>Yes Horizontal, Vertical &amp; Mixed</td>
<td>No</td>
<td>Yes Horizontal</td>
<td>Yes Horizontal, Vertical &amp; Mixed</td>
<td>Yes</td>
</tr>
<tr>
<td>Transparency Assurance</td>
<td>Yes</td>
<td>—</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Data Replication Allowance</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Transparency Assurance</td>
<td>Yes</td>
<td>—</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Data Location Transparency Assurance</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Concurrency Control Mechanism</td>
<td>TO</td>
<td>2PL Distributed Detection</td>
<td>2PL Centralised Detection</td>
<td>2PL 1 Lock Avoidance</td>
<td>2PL WaitForGraph Prevention</td>
</tr>
<tr>
<td>Deadlock Treatment</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Recovery Multiple Sites Mechanism</td>
<td>4PC</td>
<td>2PC</td>
<td>2PC</td>
<td>2PC</td>
<td>2PC</td>
</tr>
<tr>
<td>Update Distribution</td>
<td>Write All</td>
<td>—</td>
<td>Write Primary</td>
<td>Write Primary</td>
<td>Write All</td>
</tr>
<tr>
<td>System Type</td>
<td>Research</td>
<td>Research</td>
<td>Research</td>
<td>Research</td>
<td>Research</td>
</tr>
<tr>
<td>System Company</td>
<td>CCA</td>
<td>IBM</td>
<td>University of California at Berkeley</td>
<td>University of Stuttgart</td>
<td>Indiana</td>
</tr>
</tbody>
</table>

Table 3.2. A list of some Non-Long-Haul Network or Non-Homogeneous Systems.

<table>
<thead>
<tr>
<th>Group</th>
<th>Name</th>
<th>Company</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Long-Haul Network &amp; Heterogeneous System</td>
<td>DDTS</td>
<td>Honeywell</td>
<td>Research</td>
</tr>
<tr>
<td>Local Area Network System</td>
<td>Encompass</td>
<td>Tandem</td>
<td>Commercial</td>
</tr>
</tbody>
</table>

Two papers describing SDD-1 are [Roth77a] and [Roth80]. Concurrency control in SDD-1 is covered by [Bern78], [Bern80c], [Bern80b] and [McLe81], recovery by [Hamm80] and query optimization by [Wong77] and [Bern81c].

A paper introducing R* is [Wil182]. Transaction management in R* is covered by [Lind84], recovery by [Moha83], query optimization by [Sel180] and [Dani82], allocation optimization by [Lind81] and distributed control by [Lind80b].
Three papers discussing Distributed Ingres are [Ston77], [Ston80a] and [Ston80b]. Concurrency control and recovery in Distributed Ingres are covered by [Ston79] and query optimization by [Epst78].

A description of Porel may be found in [Neuh82]. Transaction management in Porel is covered by [Walt82].

An introduction to Sirius-Delta may be found in [Litw82]. A heterogeneous version of Sirius-Delta is discussed in [Ferr82].

DDTS is described in [Devo80].

Encompass is introduced in [Borr81].
Part III
4 Classes of Serializability

Here is our unifying collocation of results from several theoretical works in the serializability field. Two of them, [Bern79b] and [Papa79], have been very influential in this area.
4.1 Transaction Model

The transaction model used so far was presented in Section 2.2.4, see Eqs. 2.1 to 2.9. In [Bern79b] and [Papa79] a slightly different transaction model was employed, and we will start by pointing out the differences between this model and the common one. Further another notation was used, but we will transform the specifications to the framework already introduced.

The system database definition (Eq. 2.1) remains the same.

**System database** is a Set \( D \):

- \( D = \{x | x \text{ item}\} \) \hspace{1cm} (Eq. 4.1)

But the transaction schedule definition (Eqs. 2.2 to 2.5) changes a lot.

**Transaction schedule** \( T_i \) is a Totally Ordered Set \( (t_i, \prec_i) \):

- \( r_i \in D \) \hspace{1cm} (Eq. 4.2)
- \( w_i \in D \) \hspace{1cm} (Eq. 4.3)
- \( t_i = \{R_i(r_i), W_i(w_i)\} \) \hspace{1cm} (Eq. 4.4)
- \( R_i(r_i) \prec_i W_i(w_i) \) \hspace{1cm} (Eq. 4.5)

Eq. 4.4 shows that each transaction consists of exactly two operational elements, one read-action and one write-action. However as indicated in Eqs. 4.2 and 4.3, each of the two actions may access zero, one or more items - giving respectively the read-set and write-set. Eq. 4.5 requires that the two actions must always be ordered - with the read being followed by the write. Thus a transaction necessarily constitutes a total order - with a sequential enumeration as the natural illustration.

Note that there is no notion of a commit or abort. Observe also that the read-before-write case corresponds to that necessarily

\[ \forall i \ [r_i \triangleright w_i], \]

while the not-read-before-write case corresponds to that possibly

\[ \exists i \ [r_i \triangleright w_i]. \]

The conflicting operations definition (Eq. 2.6) now works on read-sets and write-sets.
Conflicting operations is a Binary Relation ~:

- \( \forall i, j \left[ [r_i \cap w_j \neq \emptyset \land i \neq j] \Rightarrow R_i(r_i) \sim W_j(w_j) \right] \) \hspace{1cm} (Eq. 4.6)

- \( \forall i, j \left[ [w_i \cap r_j \neq \emptyset \land i \neq j] \Rightarrow W_i(w_i) \sim R_j(r_j) \right] \) \hspace{1cm} (Eq. 4.7)

- \( \forall i, j \left[ [w_i \cap w_j \neq \emptyset \land i \neq j] \Rightarrow W_i(w_i) \sim W_j(w_j) \right] \) \hspace{1cm} (Eq. 4.8)

The complete system schedule definition (Eqs. 2.7 to 2.9) now changes into a system schedule definition.

System schedule \( H \) is a Totally Ordered Set \((h, \prec)\):

- \( h = \bigcup_{i=1}^{n} t_i \) \hspace{1cm} (Eq. 4.9)

- \( \prec \supseteq \bigcup_{i=1}^{n} \prec_i \) \hspace{1cm} (Eq. 4.10)

- \( \forall p \in t_i, q \in t_j \left[ i \neq j \Rightarrow [p \prec q \lor q \prec p] \right] \) \hspace{1cm} (Eq. 4.11)

Eqs. 4.9 and 4.10 remain the same as before, while Eq. 4.11 shows that all operational elements must be ordered. Thus also a schedule necessarily constitutes a total order - with a sequential enumeration as the natural illustration.

Note that there is no notion of a committed projection. As the projection definition and the two concatenation definitions given in Section 2.2.6 will remain the same, they will not be repeated here.

To sum up, we may say that we have moved from a multi-action-single-item model (a read- or write-action may access only one item, but we may have several such actions) to a single-action-multi-item model (a read- or write-action may access several items, but we may have only one such action). As transactions and schedules in the last model may be simulated by the first model, but not vice versa, the last model actually is a submodel of the first.

As mentioned above, the read-set or write-set of a transaction may be empty - but not both at the same time. Still we treat an empty action as a non-empty one, as this simplifies the coming specifications. However in the coming examples we leave an empty action out, as this makes them smaller. A missing read is considered to occur immediately before its corresponding (existing) write, while a missing write is considered to occur immediately after its corresponding (existing) read. The termination of a transaction may be considered to coincide with the write-action (or the read-action if the other is missing).
Let us look at some examples. They serve both as illustrations for this section and as cases to be used in the remaining parts of this chapter.

\[ H_1 = \]
\[ T_1: \quad R_1(x) \quad W_1(x) \]
\[ T_2: \quad R_2(x) \quad W_2(x) \]

\[ H_2 = \]
\[ T_1: \quad R_1(x)W_1(x) \]
\[ T_2: \quad R_2(x)W_2(x) \]

\[ H_3 = \]
\[ T_1: \quad W_1(x,y) \]
\[ T_2: \quad R_2(x) \quad W_2(v) \]
\[ T_3: \quad R_3(y) \quad W_3(x,z) \]
\[ T_4: \quad W_4(x) \]

\[ H_4 = \]
\[ T_1: \quad R_1(x) \quad W_1(y) \]
\[ T_2: \quad W_2(x) \]
\[ T_3: \quad R_3(y) \]

\[ H_5 = \]
\[ T_1: \quad R_1(x) \quad W_1(y) \]
\[ T_2: \quad W_2(x) \]
\[ T_3: \quad W_3(y,z) \]

\[ H_6 = \]
\[ T_1: \quad R_1(x) \quad W_1(y) \]
\[ T_2: \quad W_2(x) \]
\[ T_3: \quad R_3(y) \quad W_3(y) \]

\[ H_7 = \]
\[ T_1: \quad R_1(x) \quad W_1(y) \]
\[ T_2: \quad W_2(x,z) \]
\[ T_3: \quad W_3(y,z) \]

\[ H_8 = \]
\[ T_1: \quad R_1(x) \quad W_1(y) \]
\[ T_2: \quad W_2(x) \]
\[ T_3: \quad W_3(x) \]

\[ H_{10} = \]
\[ T_1: \quad R_1(x) \quad W_1(x) \]
\[ T_2: \quad R_2(y) \quad W_2(y) \]

(The examples illustrate that the set-formation symbols on the items read or written will be left out).
4.2 Classes of Schedules

In this section we will analyze and relate a set of classes discussed partly in [Bern79b] and partly in [Papa79]. We will also relate them, where appropriate, to the classes from [Bern87b] and [Papa86] already treated or mentioned in Chapter 2.

4.2.1 Class A

Here we will specify a class that was called all-logs in [Bern79b] and H in [Papa79].

All schedules is a Set A:

\[ H \in A \text{ iff } \forall i [H[\{T_i\}] = T_i] \]  

(Eq. 4.12)

This class should constrain all possible schedules to all actual schedules; i.e. where a transaction always reads its input before it writes its output. Class membership thus required that the system schedule maintained the sequences of the read- and write-actions of each single transaction schedule. As can be seen from Eq. 4.5 in the previous section, this restriction is included in our formulation of the new transaction model. But as the two source-materials used another notation, the restriction was not explicit there.

As an example, Fig. 4.1 "shows" that:

\[ H_1 \in A \]

\[ H_1 \downarrow \]

\[ H_1[T_1]=T_1 \]

&

\[ H_1[T_2]=T_2 \]

Fig. 4.1. A-test of H_1.

(The figure illustrates that the set-formation symbols on the transactions projected upon will be left out).
4.2.2 Class S

Now, for the sake of completeness, we will "repeat" the definition of the serial class from Section 2.2.6 (see Eq. 2.11). In [Bern79b] this class is not named explicitly, but in [Papa79] it is called S. Both [Bern87b] and [Papa86] name the class serial.

Serial schedules is a Set S:

\[ H \in S \text{ iff } H = T_i \odot T_j \odot \ldots T_k \]

(Eq. 4.13)

for \(i, j, \ldots, k\) being some or another Permutation of \(\{1, 2, \ldots, n\}\)

This class will contain all schedules where each transaction is executed to its end before another is begun. Hence each schedule describes a completely serial execution of its transactions. Class membership thus means that a schedule allows no interleaving of any two transactions.

As an example, Fig. 4.2 shows that:

\[ H_2 \in S \]

![Fig. 4.2. S-test of \(H_2\).]

But as another example, Fig. 4.3 shows that:

\[ H_{10} \notin S \]

![Fig. 4.3. S-test of \(H_{10}\).]
4.2.3 **CLASS FSR**

Here we will define a weaker version of the view serializable class from Section 2.2.6 (see Eqs. 2.18 to 2.33) and specify it according to the new transaction model. In both [Bern79b] and [Papa79] this class is called SR. Only [Papa86] - and not [Bern87b] - treats the class and names it final-state serializable.

Like with VSR in Section 2.2.6 we will make use of the extended version $H^*$ of a system schedule $H$:

$$ H^* = T_w \cup H \cup T_r $$  \hspace{1cm} (Eq. 4.14)

The extended schedule includes an initialization of a database state before the database is used, and a checking of the database state after it is used:

- $T_w = W_w(D)$  \hspace{1cm} (Eq. 4.15)
- $T_r = R_r(D)$  \hspace{1cm} (Eq. 4.16)

This extended schedule construct will also be used for a class called SSR in the next section.

As we will employ the alternative way of specifying schedule equivalence from Section 2.2.6 - i.e. without the use of writes-x-from notion, we only need the reads-x-from notion:

$T_i$ reads-x-from $T_j$ in $H$ iff

- $W_j(w_j) < R_j(r_i)$ $\land$ $x \in w_j \cap r_i$  \hspace{1cm} (Eq. 4.17)
  and
- $\forall w_k(w_k) \in h \left[ W_j(w_j) < W_k(w_k) < R_1(r_i) \Rightarrow x \not\in w_k \right]$  \hspace{1cm} (Eq. 4.18)

This concept is illustrated in Fig. 4.4.

We also need to know the set of transactions that directly or indirectly effect the final database state (given any initial database state). These transactions are the live ones, and an algorithm for computing them is included in Fig. 4.5. Observe that we base the computation on $H^*$ and not $H$.

All other transactions are the dead ones; i.e. they have no effect at all on the resulting values in the database.

In contrast to the requirements of (old) view equivalence as stated in Section 2.2.6, final-state equivalence only means that whatever values that the database items have initially and whichever transformations
of values read or entered into values written or printed that the transactions represent, the values that the database items finally have are to be the same for the two executions.

Final-state equivalent schedules is a Binary Relation \(=_{f}\):

\[
H =_{f} H' \text{ iff}
\]

- \(h = h'\) \hspace{1cm} (Eq. 4.19)
- \(\text{live-transactions}(H^*) = \text{live-transactions}(H'^*)\) \hspace{1cm} (Eq. 4.20)

and

- \(\forall x \in D, T_i, T_j\)
  \[
  \text{Live } T_i \text{ reads-x-from } T_j \text{ in } H^* \iff \text{Live } T_i \text{ reads-x-from } T_j \text{ in } H'^*
  \]

Final-state serializable schedules is a Set FSR:

\[
H \in \text{FSR iff}
\]

- \(\exists H_s \in S [H =_{f} H_s]\) \hspace{1cm} (Eq. 4.22)
This class constrains all actual schedules to all schedules final-state equivalent to some or another serial schedule. It is thus the class originally considered to correspond to all correct executions.

Comparing Eq. 4.21 with Eq. 2.33, we see that final-state equivalence requires the preservation of reads-item-from relations for live transactions only, while view equivalence requires the preservation for all (live and dead) transactions. Thus from comparing Eq. 4.22 with Eq. 2.28, we see that apart from the transaction model difference FSR is a wider class than VSR₀. This will be further discussed in Section 4.3. The FSR-class will also be fully specified in the Appendix for the transaction model used in Section 2.2.6 in connection with the treatment of a flaw in some reference-material.

Thus to test class membership in FSR for a given system schedule H, construct the extended version H* of H and make a directed graph-set FSG(H*) as:

\[ V(\text{FSG}(H*)) = \{T_w, T_r\} \cup \{T_i | t_i \in h\} \quad (\text{Eq. 4.23}) \]

\[ A(\text{FSG}(H*)) = \quad (\text{Eq. 4.24}) \]

\[ \{T_w \rightarrow T_i | t_i \in h\} \quad (a) \]

\[ \cup \{T_i \rightarrow T_r | t_i \in h\} \quad (b) \]

\[ \cup \{T_j \rightarrow T_i\} \quad (c) \]

\[ \exists x \in D [\text{Live } T_i \text{ reads-}x\text{-from } T_j \text{ in } H^*] \]

\[ \cup \{T_k \rightarrow T_j \lor T_i \rightarrow T_k\} \quad (d) \]

\[ \exists x \in D [\text{Live } T_i \text{ reads-}x\text{-from } T_j \text{ in } H^* \land \]

\[ T_k \text{ writes-}x\text{-also in } H^*] \]

Then \( H \in \text{FSR} \) if and only if there is at least one directed graph \( \text{FSG}_{*}(H^*) \) among the directed graph-set \( \text{FSG}(H^*) \) that is acyclic.

Case c) was basically illustrated in Fig. 4.4. But observe that \( T_i \) has to be live in case c) for this type of arc to be included. Thus after including all reads-item-from relations in computing the set of live transactions - as shown in Fig. 4.5, we will exclude some of them in testing for class membership - as shown in Fig. 4.6. Case d) is thoroughly illustrated in Fig. 4.7. Cases a) and b) respectively concern the initialization and checking of the database, and such obvious arcs will be left out in the coming examples. (This will also be done in the examples concerning the next class, SSR).

Note further that any topological sort of an acyclic \( \text{FSG}_{*}(H^*) \) corresponds to a serial schedule \( H_s \) final-state equivalent to \( H \). Thus the given system schedule \( H \) may have several final-state serializations.
Observe that apart from the transaction model difference, FSG(H*) contains less (or the same number of) arcs and arc-pairs than VSG(H*) - the serialization graph-set of VSR. This stems from the distinction between live transactions only and all (live and dead) transactions.

From the acyclic graph in Fig. 4.8 we may conclude that:

\[ H_3 \in FSR \land H_3 \equiv T_3 \circ T_1 \circ T_2 \circ T_4 \]

In this case we had to choose the alternative arc \( T_3 \rightarrow T_1 \) from the alternative arc-pair

\[ T_3 \rightarrow T_1 \lor T_2 \rightarrow T_3, \]

and we had to choose the alternative arc \( T_2 \rightarrow T_4 \) from the alternative arc-pair

\[ T_4 \rightarrow T_1 \lor T_2 \rightarrow T_4. \]

But from a cyclic graph in Fig. 4.9 we may deduce that:

\[ H_1 \notin FSR \]
Fig. 4.8. FSG($H_3^*$).

Fig. 4.9. FSG($H_1^*$).
4.2.4 **CLASS SSR**

Now we will define a strict version of class FSR from the previous section. In both [Bern79b] and [Papa79] this class is called SSR.

**Strictly serializable schedules** is a Set SSR:

\[ H \in \text{SSR} \text{ iff } \]

\[ \exists H_s \in S \left[ H \equiv_H H_s \land \right. \]

\[ \forall T_i, T_j \left[ W_j(w_j) <_h R_i(r_i) \iff W_j(w_j) <_{H_s} R_i(r_i) \right] \]

(Eq. 4.25)

This class contains all schedules final-state equivalent to some or another serial schedule without having to interchange two originally non-interleaved transactions. Thus it represents a fairly intuitive and natural extension of what should correspond to correct executions.

To test class membership in SSR for a given system schedule \( H \), construct the extended version \( H^* \) of \( H \) and make a **directed graph-set SSG(H*)** as:

- \[ V(\text{SSG}(H^*)) = \{T_w, T_r\} \cup \{T_i \mid t_i \in h\} \]  
  (Eq. 4.26)

- \[ A(\text{SSG}(H^*)) = \]  
  (Eq. 4.27)

\[ \{T_w \rightarrow T_i \mid t_i \in h\} \]  
(a)

\[ \cup \{T_i \rightarrow T_r \mid t_i \in h\} \]  
(b)

\[ \cup \{T_j \rightarrow T_i \mid W_j(w_j) <_h R_i(r_i)\} \]  
(c)

\[ \cup \{T_k \rightarrow T_j \lor T_i \rightarrow T_k\} \]  
(d)

\[ \exists x \in D \left[ \text{Live } T_i \text{ reads-x-from } T_j \text{ in } H^* \land \right. \]

\[ T_k \text{ writes-x-also in } H^* \right]\)

Then \( H \in \text{SSR} \) if and only if there is at least one directed graph SSG\(_s\)(H\(^*\)) among the directed graph-set SSG(H\(^*\)) that is acyclic.

**Case c)** - the only differing case from class FSR - is illustrated in Fig. 4.10.

We see that SSG(H\(^*\)) contains more (or the same number of) arcs than FSG(H\(^*\)) - the serialization graph-set of FSR. Hence SSG corresponds to more (or the same number of) constraints than FSG, never less. This leads to less chance of finding an acyclic test graph (in case of more arcs).
From the acyclic graph in Fig. 4.11 we may conclude that:

\[ H_4 \in \text{SSR} \land H_4 =_{_{f}} T_1 \circ T_2 \circ T_3 \]

But from a cyclic graph in Fig. 4.12 we may deduce that:

\[ H_5 \notin \text{SSR} \]
4.2.5 **CLASS CSR**

Here we will respecify the conflict serializable class from Section 2.2.6 (see Eqs. 2.12 to 2.14) according to the new transaction model. In [Bern79b] this class is called CPSR, and in [Papa79] DSR. Both [Bern87b] and [Papa86] treat the class and name it conflict serializable.

**Conflict equivalent schedules** is a Binary Relation \( =_c \):

\[
H =_c H' \ 	ext{iff} \\
- h = h' \\
\text{and} \\
- \forall p \in t_i, q \in t_j \\
[[p <_h q \land p \sim q] \rightarrow p <_h' q] \\
\]

(Eq. 4.28)

(Eq. 4.29)

**Conflict serializable schedules** is a Set CSR:

\[
H \in \text{CSR} \text{ iff} \\
- \exists H_s \in S [H =_c H_s] \\
\]

(Eq. 4.30)

In trying to find an equivalent serial schedule, this class does not allow two conflicting operations to be interchanged. Hence the corresponding schedules are conflict-preserving serializable.

We may also characterize this class by saying that the given system schedule \( H \in \text{CSR} \) if and only if there is a set

\[
\{E_t | i=1,\ldots,n\} \\
\]

so that:

\[
- \ \
\forall i, j \ [[W_j(w_j) <_h R_i(r_i) \land W_j(w_j) \sim R_i(r_i)] \rightarrow E_j < E_i] \quad (a) \\
\forall i, j \ [[R_j(r_j) <_h W_i(w_i) \land R_j(r_j) \sim W_i(w_i)] \rightarrow E_j < E_i] \quad (b) \\
\forall i, j \ [[W_j(w_j) <_h W_i(w_i) \land W_j(w_j) \sim W_i(w_i)] \rightarrow E_j < E_i] \quad (c) \\
\]

The \( E_t \)-set will constitute a possible execution sequence of such a corresponding conflict-preserving serialization.

Thus to test class membership in CSR for a given system schedule \( H \), make a **directed graph** \( \text{CSG}(H) \) as:
\[-V(CSG(H)) = \{T_i | t_i \in h\}\]  \hspace{1cm} (Eq. 4.32)
\[-A(CSG(H)) = \hspace{1cm} (Eq. 4.33)
\[
\{T_j \rightarrow T_i | W_j(w_j) \prec_h R_i(r_i) \land W_j(w_j) \sim R_i(r_i)\}\] \hspace{1cm} (a)
\[
U \{T_j \rightarrow T_i | R_j(r_j) \prec_h W_i(w_i) \land R_j(r_j) \sim W_i(w_i)\}\] \hspace{1cm} (b)
\[
U \{T_j \rightarrow T_i | W_j(w_j) \prec_h W_i(w_i) \land W_j(w_j) \sim W_i(w_i)\}\] \hspace{1cm} (c)

Then \(H \in CSR\) if and only if \(CSG(H)\) is acyclic.

Cases a), b) and c) are all illustrated in Fig. 4.13. Observe that transitive arcs stemming from case c) between two and two transactions will be left out in the coming examples. (This will also be done in the examples concerning the next class, WPL).

![Fig. 4.13. Resulting contributions to CSG(H) from different types of conflicts in H.](image)

Again note further that any topological sort of an acyclic \(CSG(H)\) corresponds to a serial schedule \(H_s\) conflict equivalent to \(H\). Thus the given system schedule \(H\) may once more have several conflict serializations.

Observe that apart from \(T_w\) plus \(T_r\) and the live/dead distinction, \(CSG(H) \in FSG(H^*)\) - the serialization graph-set of FSR. It means that one specific \(FSG_n(H^*)\) has to be acyclic. This induces a faster test, but also less chance of a positive result.

From the acyclic graph in Fig. 4.14 we may conclude that:

\[H_5 \in CSR \land H_5 \cong_c T_3 \diamond T_1 \diamond T_2\]

But from a cyclic graph in Fig. 4.15 we may deduce that:

\[H_3 \notin CSR\]
Fig. 4.14. CSG (H₅).

Fig. 4.15. CSG (H₃).
4.2.6 Class WPL

Now we will define the strict version of class CSR from the previous section. In [Bern79b] this class is called W2PL, and in [Papa79] Q.

Weakly 2phase locked schedules is a set WPL:

\[ H \in WPL \iff \exists H_s \in S [H =_c H_s \land \forall T_i, T_j [W_j(w_j) <_h R_i(r_i) \implies W_j(w_j) < H_s R_i(r_i)]] \]  
\( \text{(Eq. 4.34)} \)

In trying to find a conflict equivalent serial schedule, this class does not allow two originally non-interleaved transactions to be interchanged. Hence the corresponding schedules are strictly conflict-preserving serializable.

We may also characterize this class by saying that the given system schedule \( H \in WPL \) if and only if there is a set 

\[ \{E_i | i=1,...,n\} \]

so that:

\[ \forall i [R_i(r_i) <_h E_i <_h W_i(w_i)] \]
\( \text{(a)} \)

\[ \forall i, j [[R_j(r_j) <_h W_i(w_i) \land R_j(r_j) \not< W_i(w_i)] \implies E_j <_h E_i] \]
\( \text{(b)} \)

\[ \forall i, j [[W_j(w_j) <_h W_i(w_i) \land W_j(w_j) \not< W_i(w_i)] \implies E_j <_h E_i] \]
\( \text{(c)} \)

The \( E_i \)-set will here constitute possible instant execution times of such a corresponding weakly 2phase locked serialization. Observe thus the difference in case a) compared to the previous section (see Eq. 4.31).

In effect, a weakly 2phase locked schedule has the following characteristics:

\[ R_i(r_i) \ldots E_i \text{ is write-free in read-set } r_i: \]
\[ \forall W_j(w_j) [R_i(r_i) <_h W_j(w_j) <_h E_i \implies r_i \cap w_j = \emptyset] \]
\( \text{(a)} \)

\[ E_i \ldots W_i(w_i) \text{ is read-free in write-set } w_i: \]
\[ \forall R_j(r_j) [E_i <_h R_j(r_j) <_h W_i(w_i) \implies r_j \cap w_i = \emptyset] \]
\( \text{(b)} \)
\[ E_i \ldots W_i(w_i) \text{ is weakly write-free in write-set } w_i: \]
\[ \forall W_j(w_j) \left[ E_i \triangleleft_h E_j \triangleleft_h W_j(w_j) \triangleleft_h W_i(w_i) \Rightarrow w_j \cap w_i = \emptyset \right] \quad (c) \]

To test class membership in WPL for a given system schedule \( H \), make a directed graph \( \text{WSG}(H) \) as:

- \( V(\text{WSG}(H)) = \{ T_i | t_i \in h \} \) \hspace{1cm} (Eq. 4.37)
- \( A(\text{WSG}(H)) = \)
  \[ \{ T_j \longrightarrow T_i | W_j(w_j) \triangleleft_h R_i(r_i) \} \quad (a) \]
  \[ \cup \{ T_j \longrightarrow T_i | R_j(r_j) \triangleleft_h W_i(w_i) \wedge R_j(r_j) \sim W_i(w_i) \} \quad (b) \]
  \[ \cup \{ T_j \longrightarrow T_i | W_j(w_j) \triangleleft_h W_i(w_i) \wedge W_j(w_j) \sim W_i(w_i) \} \quad (c) \]

Then \( H \in \text{WPL} \) if and only if \( \text{WSG}(H) \) is acyclic.

Case a) - the only differing case from class CSR - is illustrated in Fig. 4.16.

![Fig. 4.16. "Definition" of \( T_i \) Follows \( T_j \) in \( H \) with resulting contribution of type a) to \( \text{WSG}(H) \).](image)

We see that \( A(\text{WSG}(H)) \supseteq A(\text{CSR}(H)) \) - the arc-set of the serialization graph of CSR. Hence \( \text{WSG} \) corresponds to more (or the same number of) constraints than \( \text{CSR} \), never less. This leads to even lesser chance of finding the test graph acyclic (in case of more arcs).

From the acyclic graph in Fig. 4.17 we may conclude that:

\[ H_6 \in \text{WPL} \quad \land \quad H_6 =_{SC} T_3 \circ T_1 \circ T_2 \]

![Fig. 4.17. WSG \( (H_6) \).](image)
But from a cyclic graph in Fig. 4.18 we may deduce that:

$H_4 \notin WPL$

Fig. 4.18. WSG ($H_4$).
4.2.7 **Class 2PL**

Now we will define a stronger version of class WPL from the previous section. In both [Bern79b] and [Papa79] this class is called 2PL.

**2phase locked schedules** is a set 2PL.

This set corresponds to those schedules that may be achieved with the 2PhaseLocking mechanism introduced in Section 2.2.7 and discussed in Section 2.3.2.

We may *characterize* this class by saying that a given system schedule $H \in \text{2PL}$ if and only if there is a set

$$\{E_i | i = 1, \ldots, n\}$$

so that:

$$\forall i \ [R_i(r_i) <_h E_i <_h W_i(w_i)] \quad (a)$$

$$\forall i, j \ [[R_j(r_j) <_h W_i(w_i) \land R_j(r_j) <_h W_i(w_i)] \Rightarrow E_j <_h E_i] \quad (b)$$

$$\forall i, j \ [[W_j(w_j) <_h W_i(w_i) \land W_j(w_j) <_h W_i(w_i)] \Rightarrow E_j <_h E_i] \quad (c)$$

The $E_i$-set will again constitute possible *instant execution times* - and necessarily also possible locked-points - of such a corresponding 2phase locked serialization. Observe thus the difference in case c) compared to the previous section (see Eq. 4.35).

In effect, a 2phase locked schedule has the following *characteristics*:

$$\forall r_i(r_i) \ldots E_i \text{ is write-free in read-set } r_i:$$

$$\forall W_j(w_j) \ [R_i(r_i) <_h W_j(w_j) <_h E_i \Rightarrow r_i \cap w_j = \emptyset] \quad (a)$$

$$\forall E_i \ldots W_i(w_i) \text{ is read-free in write-set } w_i:$$

$$\forall R_j(r_j) \ [E_i <_h R_j(r_j) <_h W_i(w_i) \Rightarrow r_j \cap w_i = \emptyset] \quad (b)$$

$$\forall E_i \ldots W_i(w_i) \text{ is write-free in write-set } w_i:$$

$$\forall W_j(w_j) \ [E_i <_h W_j(w_j) <_h W_i(w_i) \Rightarrow w_j \cap w_i = \emptyset] \quad (c)$$

Observe again the difference in case c) compared to the previous section (see Eq. 4.36).
To test class membership in 2PL for a given system schedule \( H \), construct an extended version \( H^+ \) of \( H \) by, after each of the \( n \) write-actions corresponding to an original transaction, inserting a new transaction reading no items and writing the same items as the original one:

\[
H: \quad \ldots \; W_i(w_i) \ldots \quad \text{(Eq. 4.41)}
\]

\[
H^+: \quad \ldots \; W_i(w_i)R_{n+i}(\cdot)W_{n+i}(w_i) \ldots
\]

Then \( H \in 2PL \) if and only if \( H^+ \in WPL \); i.e. if and only if \( WSG(H^+) \) is acyclic.

Obviously \( A(WSG(H^+)) \supset A(WSG(H)) \); i.e. constituting more arcs. Once more this indicates lesser chance of finding the test graph of 2PL acyclic than that of WPL.

From Fig. 4.19 we may conclude that:

\[
H_7 \in 2PL \land H_7 = T_1 \circ T_2 \circ T_3
\]

![Fig. 4.19. 2PL-test of \( H_7 \).]

(In figures like this the \( P_i \)s indicate possible locked-points).

But from Fig. 4.20 we may deduce that:

\[
H_6 \notin 2PL
\]

![Fig. 4.20. 2PL-test of \( H_6 \).]
4.2.8 Class CGS

Here we will specify a class that was called CG-secure in [Bern79b].

On our way we need to extract the operational elements of a system schedule:

**Structure**

\[ \text{Struct}[H] \]

of a system schedule \( H \) is the set of its actions

\[- = h \quad \text{(Eq. 4.42)}\]

A structure differs from the corresponding schedule by lacking the ordering pairs. It may thus be specified in advance; i.e. prior to execution. This only requires that the read- and write-sets of the transactions are given.

Based on such a priori knowledge we may construct a graph-tool for analyzing possible conflicts during execution.

**Conflict-graph**

\[ \text{CG}[\text{Struct}[H]] \]

of a structure \( \text{Struct}[H] \) is a non-directed graph on the set of its actions:

\[
- V(\text{CG}[\text{Struct}[H]]) = \{ R_i(r_i) | t_i \in h \} \cup \{ W_i(w_i) | t_i \in h \} \quad \text{(Eq. 4.43)} \\
- E(\text{CG}[\text{Struct}[H]]) = \\
\quad \{ R_i(r_i) \rightarrow W_i(w_i) | t_i \in h \} \quad \text{(a)} \\
\quad \cup \{ W_j(w_j) \rightarrow R_i(r_i) | W_j(w_j) - R_i(r_i) \} \quad \text{(b)} \\
\quad \cup \{ R_j(r_j) \rightarrow W_i(w_i) | R_j(r_j) - W_i(w_i) \} \quad \text{(c)} \\
\quad \cup \{ W_j(w_j) \rightarrow W_i(w_i) | W_j(w_j) - W_i(w_i) \} \quad \text{(d)}
\]

This is illustrated in Fig. 4.21.

The conflict-graph is a non-directed version of the (directed) serialization graph of CSR of any corresponding schedule, and the construct will specifically be used for two classes called P3P and SDD in the next two sections.

In a conflict-graph there may or may not be one or more cycles. The transactions associated with a specific cycle must be identified:
Cycle-set

Cycle[CG[Struct[H]]]_a

of a conflict-graph CG[Struct[H]] is a set of transactions

- \( = \{T_i\} \) \hspace{1cm} (Eq. 4.45)

- \( R_i(r_i) \) Is-a-Vertex-on Cycle_a in CG[Struct[H]] \( \lor \)

- \( W_i(w_i) \) Is-a-Vertex-on Cycle_a in CG[Struct[H]]

The projection of a system schedule with regard to the transactions each with at least one of its actions as vertex on a specific cycle in the corresponding conflict-graph, may be expressed:

- \( H[\text{Cycle}[\text{CG}[[\text{Struct}[H]]]]_a] \) \hspace{1cm} (Eq. 4.46)

After this introduction we may formulate the class specification.

**Conflict-graph secure schedules** is a Set CGS:

\( H \in \text{CGS} \) iff

- \( \forall \text{Cycle}[\text{CG}[[\text{Struct}[H]]]]_a \) \hspace{1cm} (Eq. 4.47)

- \( [H[\text{Cycle}[\text{CG}[[\text{Struct}[H]]]]_a] \in \text{CSR}] \)
This is an implementation-oriented subclass (with regard to assuring membership) of class CSR from Section 4.2.5. If a conflict-graph has no cycles, then no treatment is needed for any transaction in any execution of a corresponding schedule. And if a conflict-graph has one or more cycles, then each cycle (i.e., the corresponding transactions) may be treated separately, while the non-cyclic part (i.e., the corresponding transactions) may be left without treatment in any execution of a corresponding schedule. Not even all cycle-variants represent problems—thus needing treatment. The non-dangerous types of cycles—i.e., those that cannot lead to cycles in CSG (the serialization graph of CSR), as well as the dangerous types of cycles—i.e., those that can lead to cycles in CSG (the serialization graph of CSR), are shown in Fig. 4.22.

Fig. 4.22. Illustration of which CG-patterns may lead to problems or not. [Cycle: \( \geq 2 \) Edges].
This class allows each transaction to be synchronized only towards the other transactions with which it constitutes a single cycle, and each cycle may be synchronized separately and possibly differently.

In the next two sections we will look at two (out of many) possible ways of synchronizing transactions on such cycles.

Let us first relate class CGS to class CSR. In [Bern79b] it is shown that

\[ \text{CGS} \subseteq \text{CSR}, \]

and it is indicated that the inclusion is proper so that:

\[ \text{CGS} \subset \text{CSR} \]

Our treatment of class CGS (as defined in this section) and classes P3P and SDD (as to be defined in the next two sections) of course depends on our interpretation of these classes being "correct".
4.2.9 **CLASS P3P**

Now we will define a class that was called P3 in [Papa79].

**P3 protocol schedules** is a set P3P:

\[ H \in \text{P3P} \text{ iff } \]

\[
\forall \text{ Cycle}[\text{CG}[\text{Struct}[H]]], i, j, k \hspace{1cm} (\text{Eq. 4.48})
\]

\[
[[R_i(r_i) \rightarrow W_j(w_j) \text{ Is-an-Edge-on Cycle}_a \text{ in CG}[\text{Struct}[H]]] \land
\]

\[
[W_i(w_i) \rightarrow R_k(r_k) \text{ Is-an-Edge-on Cycle}_a \text{ in CG}[\text{Struct}[H]]] \lor
\]

\[
W_i(w_i) \rightarrow W_k(w_k) \text{ Is-an-Edge-on Cycle}_a \text{ in CG}[\text{Struct}[H]]] \land
\]

\[
j \neq i \land k \neq i \Rightarrow
\]

\[
[W_j(w_j) \triangleleft_h R_i(r_i) \triangleleft_h W_i(w_i) \lor
\]

\[
R_i(r_i) \triangleleft_h W_i(w_i) \triangleleft_h W_j(w_j)]
\]

The types of cycle considered in this class are shown in Fig. 4.23. This exactly represents the dangerous ones - i.e. cases \( d_1 \) and \( d_2 \) from Fig. 4.22 in the previous section.

![Fig. 4.23. CG-cycle types considered in P3P.](image)

This is a general way of assuring the condition in Eq. 4.47 in the previous section, as all the considered cycles and only these may lead to cycles in CSG(H).

However note that here we force all \( H[\text{Cycle}[\text{CG}[\text{Struct}[H]]]] \) to be members of a subclass of 2PL which itself is a small subclass of CSR. Thus the method is far from general enough.

As there is no complex cycle in Fig. 4.24, we may conclude that:

\[ H_g \in \text{P3P} (\land H_g \equiv_c T_1 \square T_2 \square T_3) \]
But as there is a complex cycle in Fig. 4.25,

\[ R(x) \prec W_2(x,z) \prec W_1(y) \text{ in } H_7, \]

we may deduce that:

\[ H_7 \not\in \text{P3P} \]
4.2.10 **CLASS SDD**

Now we will define a stronger version of class P3P from the previous section. In [Bern79b] this class is called SDD-1 secure.

**SDD-1 secure schedules** is a Set SDD:

\[
H \in \text{SDD} \text{ iff } \quad (\text{Eq. 4.49})
\]

\[
\forall \text{ Cycle[CG[Struct[H]]]}_a, i, j \\
[\{R_i(r_i) \rightarrow W_j(w_j) \text{ is an Edge on Cycle}_a \text{ in } \text{CG[Struct[H]]} \land \]
\[ j \neq i \} \Rightarrow \]
\[ W_j(w_j) \prec_h R_i(r_i) \prec_h W_i(w_i) \lor \]
\[ R_i(r_i) \prec_h W_i(w_i) \prec_h W_j(w_j)]
\]

The types of cycle considered in this class are shown in Fig. 4.26. This represents the two dangerous ones - i.e. cases d1) and d2) from Fig. 4.22 in Section 4.2.8, and a non-dangerous one - i.e. case c) from the same figure.

![Fig. 4.26. CG-cycle types considered in SDD.](image)

Compared to P3P this is a special way of assuring the condition in Eq. 4.47 in Section 4.2.8, as not all the considered cycles may lead to cycles in CSG(H).

Further note that here we force all H[Cycle[CG[Struct[H]]]] to be members of a smaller subclass of 2PL which itself again is a small subclass of CSR. Thus the method is actually very special.

As there is no simple cycle in Fig. 4.27, we may conclude that:

\[
H_{10} \in \text{SDD} \land H_{10} \ast_c T_1 \ast_c T_2 \ast_c T_2 \ast_c T_1
\]
But as there is a simple cycle in Fig. 4.28,

![Diagram](image)

Fig. 4.28. CG [Struct[H₂]].

and

\[ R_1(x) < W_2(x) < W_1(y) \] in \( H_9 \),

we may deduce that:

\[ H_9 \notin \text{SDD} \]
4.3 CLASS RELATIONS AND FACTS

Initially, we want to illustrate the relations between the classes of schedules that we have analyzed in this chapter (A, FSR, SSR, CSR, WPL, 2PL, CGS, P3P, SDD and S). Before we can do that, we have to carry out a few more tests on our examples to show that we have covered all the different possibilities.

In Section 4.2.3 we could conclude that:

\[ H_3 = T_3 \circ T_2 \circ T_4 \Rightarrow H_3 \in \text{FSR} \]

We have in \( H_3 \)

\[ W_1(x,y) < R_2(x) \land W_2(v) < R_4() W_4(x) \land W_3(x,z) < R_4() W_4(x), \]

so the extra strictness requirements are:

\[ T_1 < T_2 \land T_2 < T_4 \land T_3 < T_4 \]

All these are fulfilled by the above-mentioned final-state serialization of \( H_3 \), so from this we may even deduce that:

\[ H_3 \in \text{SSR} \]

From Fig. 4.29 we may conclude that:

\[ H_9 \in \text{2PL} \land H_9 = T_1 \circ T_2 \circ T_3 \]

![Fig. 4.29. 2PL-test of \( H_9 \).]

From Fig. 4.30 we may conclude that:

\[ H_{10} \in \text{2PL} \land H_{10} = T_1 \circ T_2 \Rightarrow T_2 \circ T_1 \]

![Fig. 4.30. 2PL-test of \( H_{10} \).]
As there is no simple cycle in Fig. 4.31,

we may conclude that:

\[ H_4 \in \text{SDD} (\land H_4 = T_3 \uplus T_1 \uplus T_2) \]

As there is no simple cycle in Fig. 4.32,

we may conclude that:

\[ H_5 \in \text{SDD} (\land H_5 = T_3 \uplus T_1 \uplus T_2) \]

As there is a simple cycle in Fig. 4.33,

but

\[ R_3(y) < W_3(y) < W_1(y) \text{ in } H_6, \]

we may conclude that:

\[ H_6 \in \text{SDD} (\land H_6 = T_3 \uplus T_1 \uplus T_2) \]
First, let us give the relations between the classes of schedules analyzed for the general case of not-read-before-write (see Section 4.1). Fig. 4.34 covers this case, and here we have included example cases for the different possibilities. Compare this with the earlier Fig. 2.32 in Section 2.2.9.

Fig. 4.34: How the Classes of Schedules are related in the General Case:
I.e. \( w_i \in R_j \) for \( W_i(w) \) & \( R_j(r) \) for one or more \( i=1, \ldots, N \).

The example cases indicated in the figure without having been presented so far, may for example be constructed as simple concatenations of previous examples:

\[
\begin{align*}
H_{13} & = H_3 \cdot H_5 \\
H_{24} & = H_4 \cdot H_7 \\
H_{25} & = H_5 \cdot H_7 \\
H_{26} & = H_6 \cdot H_7 \\
H_{34} & = H_4 \cdot H_9 \\
H_{35} & = H_5 \cdot H_9 \\
H_{36} & = H_6 \cdot H_9
\end{align*}
\]
However the two crossed cases $H_{44}$ and $H_{45}$ have not been covered.

Now, let us cover the special case of read-before-write (again see Section 4.1). Fig. 4.34 here changes into Fig. 4.35. Once more compare this with the earlier Fig. 2.43 in Section 2.2.9.

![Diagram]

Fig. 4.35 How the Classes of Schedules are related in the Special Case: i.e. $w_i \subseteq r_i$ for $W_i(w_i)$ & $R_i(r_i)$ for each and all $i = 1, ... N$.

As may be seen, the immediate consequences are:

- $CSR = FSR$ (instead of $CSR \subset FSR$)
- $WPL = SSR$ (instead of $WPL \subset SSR$)

Second, let us tabulate some known and unknown facts about the different classes of schedules. Table 4.1 covers the not-read-before-write case, while Table 4.2 covers the read-before-write case.

The membership-testing column deals with the possible speed of checking whether any given schedule is a member in a specific class or not. Knowing that a certain schedule is a member in a specific class, the serializing column deals with the time required to find a corresponding equivalent serial schedule. The scheduling column deals with the time required to transform any given schedule (through reordering its operational elements) into a certain schedule member in a specific class; i.e. forcing a specific class-membership onto any given schedule.
Table 4.1. Known and unknown facts about the Classes of Schedules in the General Case where not necessarily everything to be written by a transaction first must be read by it.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Membership-Testing:</th>
<th>Serializing:</th>
<th>Scheduling:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Check a $H_A \in A$ for $H_A \in \text{Class}$</td>
<td>Find a $H_e \in S$ so that $H_e = H_e \in \text{Class}$</td>
<td>Transform a $H_A \in A$ into $H_A' \in \text{Class}$</td>
</tr>
<tr>
<td>FSR</td>
<td>N</td>
<td>-</td>
<td>N</td>
</tr>
<tr>
<td>SSR</td>
<td>?</td>
<td>-</td>
<td>?</td>
</tr>
<tr>
<td>CSR</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>CGS</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>WPL</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>2PL</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>P3P</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>SDD</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>S</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

Y means: Yes, there is a quick (polynomial time) solution to the problem
N means: No, there is no quick (polynomial time) solution to the problem
? means: It is not known whether there is a quick solution to the problem or not

Table 4.2. Known facts about the Classes of Schedules in the Special Case where necessarily everything to be written by a transaction first must be read by it.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Membership-Testing:</th>
<th>Serializing:</th>
<th>Scheduling:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Check a $H_A \in A$ for $H_A \in \text{Class}$</td>
<td>Find a $H_e \in S$ so that $H_e = H_e \in \text{Class}$</td>
<td>Transform a $H_A \in A$ into $H_A' \in \text{Class}$</td>
</tr>
<tr>
<td>FSR = CSR</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>CGS</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>SSR = WPL</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>2PL</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>P3P</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>SDD</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>S</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

Y means: Yes, there is a quick (polynomial time) solution to the problem

Now, let us tabulate the consequences that the facts given above have on the practical applicability of different scheduling mechanisms. Table 4.3 covers the not-read-before-write case, while Table 4.4 covers the read-before-write case.

(The \( \text{im} \)- and \( \text{M}^1 \)-expressions are only meant to be complexity indicators).

Like in Tables 4.1 and 4.2 scheduling - or membership-forcing - effectively means transforming any given schedule $H_A$ (member in the universal class A) into a certain schedule $H_A'$ member in a specific
Table 4.3. Cost/Benefit of different scheduling methods in the General Case.

<table>
<thead>
<tr>
<th>Scheduling (≥: Membership-Forcing)</th>
<th>Complexity</th>
<th>Usability</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSR</td>
<td>NP-Complete!</td>
<td>Inefficient Operation!</td>
</tr>
<tr>
<td>SSR</td>
<td>NP-Complete?</td>
<td>Inefficient Operation?</td>
</tr>
<tr>
<td>CSR</td>
<td></td>
<td>Advanced</td>
</tr>
<tr>
<td>CGS</td>
<td></td>
<td>Common</td>
</tr>
<tr>
<td>WPL</td>
<td></td>
<td>Inefficient Result</td>
</tr>
<tr>
<td>2PL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SDD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NP-Complete means: There is (most probably) only an exponential time solution (""\( M \)": \( M \)= no. of transactions/actions) to the scheduling problem

\( \neg \text{NP-Complete} \) means: There is a polynomial time solution (""\( M' \)": \( M' \) = no. of transactions/actions) to the scheduling problem

Table 4.4. Cost/Benefit of different scheduling methods in the Special Case.

<table>
<thead>
<tr>
<th>Scheduling (≥: Membership-Forcing)</th>
<th>Complexity</th>
<th>Usability</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>Advanced</td>
</tr>
<tr>
<td>FSR = CSR</td>
<td></td>
<td>Common</td>
</tr>
<tr>
<td>CGS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSR = WPL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2PL</td>
<td>( \neg \text{NP-Complete} )</td>
<td>Inefficient Result</td>
</tr>
<tr>
<td>P3P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SDD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( \neg \text{NP-Complete} \) means: There is a polynomial time solution (""\( M' \)": \( M' \) = no. of transactions/actions) to the scheduling problem

class C. The complexity column is a summary of the scheduling column in the two previous tables, and the effects of this with regard to on-line scheduling is given in the usability column.

In on-line systems the possible speed of scheduling is very important. Hence exponential time solutions are considered unacceptable, while polynomial time solutions are acceptable. For the general case this rules out FSR (and most probably SSR) as the goal class for a scheduling mechanism. The same would also apply to both VSR₀ and VSR₉, see below.
On the other hand, the possible amount of parallelism is as crucial in on-line systems. This rules out S as the goal class for a scheduling mechanism for both the general and special cases.

For the **general case** CSR is a practical compromise between the need for an efficient operation - with regard to the speed of scheduling, and the need for an efficient result - with regard to the amount of parallelism.

For the **special case** even SSR or FSR may be candidates, as they coincide with respectively WPL and CSR. The same would also apply to both VSR$_N$ and VSR$_O$, see below.

(In determining the appropriate scheduling mechanism, we should also take into account the trade-off between concurrency and reliability as discussed in Section 2.2.8).

Tables 4.1 to 4.4 reflect some results from a work that was initiated by [Papa77] and continued by [Papa79] among others. [Gare79] contains an introduction to NP-completeness, and [Hopc79] treats the same area from yet another angle.

Further, we wish to include a more informal discussion of the appropriateness of different classes of schedules as correctness criterion for system scheduling. This necessitates taking a closer look at the assumptions underlying both the general model in Chapter 2 and the special model in this chapter. As the general model is a supermodel of the special model - see Section 4.1, we will carry out our analysis employing the multi-action-single-item model.

As in Section 2.2.7 let us consider what kinds of information about the transactions and/or database that could be **accessible** to the system scheduler:

- **Syntax:**
  
  This deals with which items that are used - selected, and how these items are used - read or written.

  The effect of a retrieval $R_i(x)$ is to set a local variable
  \[ t_i, x_R := x, \]
  and the effect of an update $W_i(y)$ is to sense a local variable:
  \[ y := t_i, y_W \]

  Such **access information** (which and how) is supposed to be known at least dynamically; i.e. at the time of the retrieval or update.

- **Semantics:**

  This deals with which mappings that the actions of a transaction implement logically.
Focusing on straight-line computation only, a transformation from all the already retrieved values to a value to be updated, would be expressed:

\[ t, y_W := f, y_W (\{ t, x_R | R(x) < W(y) \}) \]

Such functional dependencies are considered to exist, but the corresponding information (the transforming functions themselves) is supposed to be unknown. This concerns the interpretation of transactions and schedules.

Focusing on control structure decisions too, a predicate governing whether an update is to take place or not, would be expressed:

\[ \text{Do } W(y) \text{ If } c, y_W (\{ t, x_R | R(x) < W(y) \}) \]

Such conditional dependencies may be considered to exist - added to the functional dependencies, but the corresponding information (the predicated conditions themselves) is eventually supposed to be unknown. This concerns a so-called full interpretation of transactions and schedules.

- Integrity Constraints:

This deals with how the items of a database are connected logically.

The predicates governing the assertions to be maintained for the database, would be expressed:

\[ \{ p_k, i (\{ z | z \in D \}) \} \]

Such relational dependencies are considered to exist, but the corresponding information (the asserting predicates themselves) is supposed to be unknown.

In Section 8.1 we will also analyze the use of such different kinds of knowledge from yet another angle.

First, let us look at the relationship between the classes of schedules CSR, VSR, VSR and FSR. The general versions of the first three classes were all discussed in Chapter 2, while the special version of the last class has been discussed in this chapter - with its general version included in the Appendix. In the following treatment we restrict the general versions of CSR and VSR to complete system schedules only (like VSR and FSR).

For the general multi-action-single-item model (and actually the special single-action-multi-item model too) we have the following relations for the general not-read-before-write case:

\[ \text{FSR} \supset \text{VSR} \supset \text{VSR} \supset \text{CSR} \]
We need to emphasise some concepts. The state of the database comprises the values of all the existing items, while the view of a transaction comprises the values of all its retrieved items.

Database-State: \( \{x \mid x \in D\} \)

Transaction\(_t\) View: \( \{x \mid R_t(x) \in t\} \)

A transaction may either be any normal transaction retrieving and/or updating parts of the items of the database, the extra transaction from the extended schedule checking the values of all the items in the database after execution, or a hypothetical transaction checking the values stemming from committed transactions only of all the items in the database during execution.

\( T_i \) : any normal transaction accessing database-items

\( T_r \) : the extra transaction sensing the final database-state;
   i.e. retrieving all the item-values in the end

\( T_c \) : a hypothetical transaction sensing the current database-state;
   i.e. retrieving all the item-values at any time -
   reflecting the updates of committed transactions only

We may now describe the classes of schedules.

- FSR:

For any initial database-state and any interpretation, the final database-state resulting from a schedule \( H \) must be the same as that of a corresponding serial schedule \( H_s \);

\( T_r \)-View Equal for \( H \) and \( H_s \).

- VSR\(_0\):

For any initial database-state and any interpretation, the final database-state and the separate single transaction-views resulting from a schedule \( H \) must be the same as that of a corresponding serial schedule \( H_s \);

\( T_r \)-View and \( T_i \)-Views Equal for \( H \) and \( H_s \).

- VSR\(_N\):

For any initial database-state and any interpretation, the final database-state, the separate single transaction-views and all the possible prefix-corresponding commit-projected temporary database-states resulting from a schedule \( H \) must be the same as that of a corresponding serial schedule \( H_s \);

\( T_r \)-View, \( T_i \)-Views and \( T_c \)-Views Equal for \( H \) and \( H_s \).
- CSR:

The action ordering of all conflicting operations resulting from a schedule $H$ must be the same as that of a corresponding serial schedule $H_s$.

We will then compare these classes two by two:

- FSR vs. VSR$_0$:

FSR makes a distinction between dead and live transactions, while VSR$_0$ does not - i.e. VSR$_0$ assumes that all transactions are live.

The graph-tests checking membership of the classes thus only differ in that those transactions that are found dead in FSG, are still treated as live in VSG.

(To be totally precise we have to refer to the distinction in the Appendix between live and dead actions - and not only between live and dead transactions. Thus in assuring equivalent serial schedules, FSR preserves reads-from-relations of all live read-actions - and their corresponding write-actions - in any total schedule, while VSR$_0$ preserves reads-from-relations of all live and dead read-actions - and their corresponding write-actions - in any total schedule).

As an example of the distinction between FSR and VSR$_0$, see the analysis of the schedule $H_{11}$ below. This schedule is valid in both the general and special transaction models. See also the schedule $H_{11}$ of Section 2.2.9. It is valid only in the general transaction model. Reviewing this chapter's example schedule $H_3$, we saw in Section 4.2.3 that it has no dead transactions. Thus $H_3$ must be a member in VSR$_0$ as it is a member in FSR. We also found at the beginning of this section that this schedule fulfilled the strictness requirements. Thus $H_3$ is even a member in the strict version of VSR$_0$ (in addition to SSR). But reviewing this chapter's example schedule $H_4$, we saw in Section 4.2.4 that it has some dead transactions. Actually this will indirectly make $H_4$ not being a member in the strict version of VSR$_0$, even though it is a member in both SSR and VSR$_0$.

If for example each transaction (following all its retrievals) updates a private variable, then the assumption of VSR$_0$ about all transactions (read-actions) being live is 100% appropriate, as in this case all transactions (read-actions) will be live per definition. Including the terminal screen in the analysis - as indicated in Section 2.2.6, is one way of assuring that each transaction updates a private variable - through its output message.

- VSR$_0$ vs. VSR$_N$:

VSR$_0$ is less failure oriented, while VSR$_N$ is more failure oriented. This concerns the commit-projection-closed property
that is so important in on-line scheduling, as discussed in Section 2.2.6. VSR₀ does not have this property, while VSRₙ does.

The graph-tests checking membership of the classes differ in that VSR₀ only employs VSG on the total schedule itself, while VSRₙ uses VSG on each possible committed projection of a prefix of the schedule - including the total schedule itself.

(Thus in assuring equivalent serial schedules, VSR₀ preserves reads-from-relations of all transactions in any total schedule, while VSRₙ preserves reads-from-relations of all transactions in each possible committed projection of a prefix of any total schedule).

So while each transaction being live is an assumption in VSR₀, it is a "reality" for more write-transactions in VSRₙ - i.e. more write-transactions are live in some or another (may be different) of the prefixes of the schedule being analyzed.

As an example of the distinction between VSR₀ and VSRₙ, recall the discussion of schedule H₃⁻ in Section 2.2.9. That was again a variant of this chapter's example schedule H₃.

- VSRₙ vs. CSR:

VSRₙ is more correctness oriented, while CSR is more efficiency oriented. This concerns the amount of parallelism versus the speed of scheduling that both are so important in on-line systems, as discussed earlier in this section.

The graph-tests checking membership of the classes differ in that VSG deals with reads-item-from relations, while CSG deals with conflicting operations relations.

(For a further elaboration of the differences between [FSR, VSR₀ &] VSRₙ and CSR, see the informal discussion of classes VSR and CSR at the end of Section 2.2.6).

As examples of the distinction between VSRₙ and CSR, see the schedules H₆ and H₁₀/H₁₀ of Sections 2.2.6 and 2.2.9 respectively. These schedules are valid only in the general transaction model. Recall also the discussion of schedule H₃⁻ in Section 2.2.9. That was once more a variant of this chapter's example schedule H₃.

Again we need to emphasise some concepts. A commit-projection-closed class maintains membership for each possible committed projection of a prefix of a schedule that is itself a member in the class, while a projection-closed class maintains membership for any possible projection of a schedule that is itself a member in the class.

Commit-projection-closed Class C:

\[ H_c \in C \Rightarrow \forall H'_c \left( H'_c \text{ is-a-Prefix-of } H_c \Rightarrow C(H'_c) \in C \right) \]
Projection-closed Class C:

\[ H_c \in C = H_c[\{T_i, T_j, \ldots T_k\}] \in C \]

for \( \{i, j, \ldots k\} \) being any possible Subset of \( \{1, 2, \ldots n\} \)

We will now again couple some of the classes of schedules.

- FSR and VSR_0 are not commit-projection-closed, while VSR_N and CSR are. VSR_N is the widest subclass of VSR_0 that is commit-projection-closed. Refer to Section 2.2.6.

- FSR, VSR_0 and VSR_N are not projection-closed, while CSR is. CSR is the widest subclass of VSR_0 that is projection-closed, see [Yann84].

- If all transactions (actions) are live, then FSR = VSR_0. Refer to this section.

- If "for any interpretation" (i.e. considering straight-line computations only) in the description of final-state equivalence is changed to "for any full interpretation" (i.e. considering control structured computations too), then FSR' = VSR_0, see [Papa86].

We may then again characterise these classes one by one:

- FSR:

  This is the widest class of schedules correct with regard to parallelism under any possible interpretation.

- VSR_0:

  This is the widest class of schedules correct with regard to parallelism under any possible full interpretation.

- VSR_N:

  This is the widest class of schedules correct with regard to parallelism under any (full) interpretation and tolerating system-failures in on-line systems (eventually adding RC-, ACA- or ST-restrictions addressing transaction-failures).

- CSR:

  This is the widest class of schedules correct with regard to parallelism under any (full) interpretation and tolerating system-failures in efficient on-line systems (eventually adding RC-, ACA- or ST-restrictions addressing transaction-failures).
To illustrate the difference between class FSR and class VSR₀, we will analyze the following schedule:

\[
H_{11} = \\
T_1: & W_1(x) \\
T_2: & R_2(x)W_2(y) \\
T_3: & R_3(x)W_3(y)
\]

In Fig. 4.36 the distinction between dead and live transactions is included in the analysis,

![Fig. 4.36. FSG (H_{11}).](image)

and from this we may conclude that:

\[ H_{11} \in \text{FSR} \land H_{11} \equiv T_2 \otimes T_3 \otimes T_1 \]

But in Fig. 4.37 a distinction between dead and live transactions is excluded from the analysis,

![Fig. 4.37. VSG (H_{11}).](image)

and from this we may deduce that:

\[ H_{11} \in \text{VSR}_0 \quad (\Rightarrow H_{11} \notin \text{VSR}_N) \]
Second, let us clarify the consequences of employing different combinations of models

General model: Multi-action-single-item;
the accesses (retrievals or updates) of several items occur as several separate units

Special model: Single-action-multi-item;
the access (retrieval or update) of several items occurs as one atomic unit,

and cases

General case: Not-read-before-write;
an update of an item may be occurring alone

Special case: Read-before-write;
an update of an item must be preceded some time before by a retrieval on the same item

Extreme case: Read-with-write;
an update of an item must be preceded immediately before by a retrieval on the same item - they occur together as an atomic unit.

A discussion of the six possible combinations follows.

1) Not-read-before-write Case

We may have read-only, write-only or read-write access, but never write-read access.

a) Multi-action-single-item Model:

Here we have

- FSR ∪ VSR₀ ∪ VSRₙ ∪ CSR, \hspace{1cm} \text{(Eq. 4.50)}

and this type was analyzed in [Papa77].

b) Single-action-multi-item Model:

Now we have the same results as in 1a) above, and this type was used in [Bern79b] and [Papa79].
2) Read-before-write Case

We may have read-only or read-write access, but never write-only access.

a) Multi-action-single-item Model:

Here we have

- $FSR \supset VSR_0 = VSR_N = CSR,$ \hspace{1cm} (Eq. 4.51)

and this type was the basis in [Stea76].

b) Single-action-multi-item Model:

Now we have

- $FSR = VSR_0 = VSR_N = CSR,$ \hspace{1cm} (Eq. 4.52)

and this type was discussed in [Bern79b] and [Papa79].

3) Read-with-write Case

We may never have read-action-write access.

a) Multi-action-single-item Model:

Here we have the same results as 2a) above for read-write and read-only accesses (variant I). While we have the same results as in 2b) above for read-write accesses only (variant II). This type was the basis in [Eswa76].

b) Single-action-multi-item Model:

Now we have

- $FSR = VSR_0 = VSR_N = CSR (= S).$ \hspace{1cm} (Eq. 4.53)

But this type is fairly artificial, as all the schedules will be serial per definition.

Another general and early discussion of the relationship between and consequences of different existing correctness criteria may be found in [Yann80].

Finally, we intend to investigate the correspondence between the theoretical access types - as analyzed in this section, and the practical lock modes - as treated in Section 2.3.2.
In most real systems several separate retrievals and/or updates may be applied. So we concentrate on the general multi-action-single-item model. Let us look into the different possibilities.

i) Read-Write Locks:

Whether retrieving, updating or retrieving and updating an item, one has to use a general X-lock.

This mode corresponds to the above combination 3a_{II}) - with FSR = CSR. So the information in Tables 4.2 and 4.4 is valid here. Specifically we have that FSR membership-testing or -forcing is possible in polynomial time.

The effects with regard to scheduling are illustrated in Fig. 4.38 which is a special variant of Fig. 2.18 in Section 2.2.6.

Consequently a scheduling mechanism corresponding to this mode is usable, as the efficiency is as needed.

![Diagram](image)

Fig. 4.38. Implications regarding transaction ordering in serial schedules equivalent to the given schedule using only Read-Write Locks.

ii) Read-Only + Read-Write Locks:

When retrieving an item only, one may use an S-lock. But updating or retrieving and updating an item requires the general X-lock.

This mode corresponds to the above combination 3a_{I}) - with FSR ⊃ CSR but VSR = CSR. So once more the information in Tables 4.2 and 4.4 is valid when we exchange FSR with VSR. Specifically we have that (FSR/) VSR membership-testing or -forcing still is possible in polynomial time.

The effects with regard to scheduling are illustrated in Fig. 4.39 which is a general variant of Fig. 2.18.

Consequently a scheduling mechanism corresponding to this mode is more applicable than mode i), as the efficiency is not decreasing for the increased level of detail of access.
iii) Read-Only + Write-Only Locks:

When retrieving an item only, one may use the S-lock. When updating an item only, one uses a special X-lock. But retrieving and updating an item requires both the S-lock and the special X-lock.

This mode corresponds to the above combination 1a) - with FSR \(\supset\) VSR \(\supset\) CSR. So here the information in Tables 4.1 and 4.3 is valid. Specifically we have that FSR (\(/\)VSR \(\supset\)) membership-testing or -forcing (most probably) is possible in exponential time only.

The effects with regard to scheduling are illustrated in Fig. 4.40 which is a combined variant of Figs. 2.25 and 2.26 in Section 2.2.6.

Consequently a scheduling mechanism corresponding to this mode is not advisable compared to mode ii), as the efficiency is decreasing beyond acceptance for the increased level of detail of access. To make this mode tolerable with regard to efficiency, the scheduler has to assume (wrongly) that any update is preceded by a retrieval - i.e. effectively ending up with the above combination 2a). But even this adapted mode is less desirable, as a combined retrieval and update requires two locks instead of one as in mode ii).
iv) Read-Only + Write-Only + Read-Write Locks:

This is a combination of modes iii) and ii).

Again $FSR \supseteq VSR_0 \supseteq CSR$ prevails, and once more Tables 4.1 and 4.3 are appropriate.

An illustration would combine Figs. 4.40 and 4.39.

Again such a scheduling mechanism would not be usable, once more because of efficiency reasons.
To sum up and extract a few essential notions, we return to Figs. 4.34 and 4.35 (or Figs. 2.32 and 2.43) and Tables 4.1 to 4.4.

- The more general case investigated - i.e. going from read-before-write to not-read-before-write, the more schedules does a specific class contain (e.g. focusing on FSR). And the longer time does the membership-testing or -forcing take ("M" ⇒ "M'")/"iM"), but the larger is the possibility of membership. Thus the fewer retrievals and/or updates have to wait; i.e. the higher is the parallelism of transactions.

- For a given case - i.e. focusing either on read-before-write or not-read-before-write, the more general class investigated (e.g. going from 2PL to FSR), the more schedules does it contain. And the longer time does the membership-testing or -forcing take ("M" ⇒ "M'"/"iM"), but the larger is the possibility of membership. Thus the fewer retrievals and/or updates have to wait; i.e. the higher is the parallelism of transactions.
5 LEVELS OF NON-SEMondayZABILITY

Here is our unifying collocation of results from some other theoretical works in the non-serializability field. One of them, [Gray76], corresponds to initial attempts in our direction.
5.1 TRANSACTION MODEL

In Chapter 4 when studying different ways of achieving serializable transactions, we used a slightly different transaction model than the basic one in Chapter 2. We use yet another slightly varied transaction model in this chapter, in order to study different consequences of allowing non-serializable transactions.

The model employed here includes locks explicitly, and it is illustrated in Figs. 5.1 and 5.2. The main focus is on the environment of respectively a single write-action and a single read-action.

Each transaction is initiated by issuing a non-visible transaction-start and subsequently finished by issuing a transaction-commit. The transaction-period so bracketed may contain one or more write-actions and/or one or more read-actions.

Each write-action (see Fig. 5.1) may be protected against other read- and write-actions by issuing a write-prepare (or lock-exclusive) sometimes before the write-action and issuing a write-commit (or unlock-exclusive) sometimes after the write-action. This pair of actions defines the exclusive-locked-period. The unlock-exclusive is also the transition point between the uncommitted-write-period (within
which the updated item may be written again) and the committed-write-period (within which the update is final).

Likewise, each read-action (see Fig. 5.2) may be protected against other write-actions by issuing a read-prepare (or lock-shared) sometimes before the read-action and issuing a read-commit (or unlock-shared) sometimes after the read-action. This pair of actions defines the shared-locked-period. The unlock-shared is also the transition point between the uncommitted-read-period (within which the retrieved item may be read again) and the committed-read-period (within which the retrieval is final).

The compatibility matrix for the lock variants used here is given in Fig. 5.3. As in Section 2.3.2, X means exclusive-lock, while S means shared-lock. The compatibility matrix of Fig. 2.44 is an extension of the one here. This stems from the extra treatment of the U or update-lock in Section 2.3.2.
<table>
<thead>
<tr>
<th>Requesting Lock</th>
<th>-</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>S</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>X</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

Fig. 5.3. Compatibility Matrix between an Existing Lock and a Requesting Lock.
5.2 Consequence Analysis

Let us now investigate the consequences of some different types of locks. Table 5.1 summarises the effects of 3 types of exclusive-locks, while Table 5.2 does the same for the corresponding types of shared-locks.

Table 5.1. The effects of 3 different types of Exclusive-Locks.

<table>
<thead>
<tr>
<th>X-Lock</th>
<th>Commit-Effect</th>
<th>Change-Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No Uncommitted-Write-Period</td>
<td>The Change is never Tested or Marked, but Immediately Committed</td>
</tr>
<tr>
<td>Short</td>
<td></td>
<td>The Change is Tested and Immediately Committed</td>
</tr>
<tr>
<td>Long</td>
<td>No Uncommitted-Write-Period before Transaction-Commit</td>
<td>The Change is Tested and Marked and remains Uncommitted</td>
</tr>
</tbody>
</table>

Table 5.2. The effects of 3 different types of Shared-Locks.

<table>
<thead>
<tr>
<th>S-Lock</th>
<th>Commit-Effect</th>
<th>Change-Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No Uncommitted-Read-Period</td>
<td>Any Change is never Tested or Guarded</td>
</tr>
<tr>
<td>Short</td>
<td></td>
<td>Any Change is Tested</td>
</tr>
<tr>
<td>Long</td>
<td>No Uncommitted-Read-Period before Transaction-Commit</td>
<td>Any Change is Tested and Guarded</td>
</tr>
</tbody>
</table>

Using no lock (exclusive or shared) indicates that the action (write or read) is not bracketed by a lock-unlock pair at all. A short lock (exclusive or shared) means that the item written or read is unlocked immediately after the action. Hence the commit point (write or read) coincides with the action point (see Figs. 5.1 and 5.2). The long lock (exclusive or shared) means that the item written or read is locked completely until the transaction finishes. Hence the commit point (write or read) coincides with the transaction-commit point (see Figs. 5.1 and 5.2).

Note that these definitions imply that there are no uncommitted-periods (write or read) for actions with no locks or short locks (exclusive or shared) as they are immediately committed. And there are no committed-periods (write or read) for actions with long locks (exclusive or shared) as they remain uncommitted.

The change-effect considered in Table 5.1 refers to the update made by the designated write-action itself. To test this change means to check whether the write-action is executable; i.e. that no other read- or write-action is currently holding a lock on the item. To mark the change means to announce to future read- and write-actions that a write-action has been executed; i.e. to leave an exclusive-lock on the item.

On the contrary, the change-effect considered in Table 5.2 refers to any update made by other write-actions to the same item as the designated read-action retrieves. To test such changes means to check
whether the read-action is executable; i.e. that no other write-action is currently holding a lock on the item. To guard such changes means to announce to future write-actions that a read-action has been executed; i.e. to leave a shared-lock on the item.
5.3 **CONSISTENCY HIERARCHY**

Then it is possible to specify different combinations of an exclusive-lock and a shared-lock. The set of combinations analyzed here is given in Fig. 5.4. Notice that only 4 out of 9 possibilities are used.

<table>
<thead>
<tr>
<th>X-Locks</th>
<th>S-Locks</th>
<th>No</th>
<th>Short</th>
<th>Long</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td></td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Short</td>
<td></td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Long</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>

Fig. 5.4. Used combinations of Exclusive-Locks and Shared-Locks.

What these 4 combinations - labelled 0, 1, 2 and 3 - mean from an implementational point of view, is respecified in Table 5.3.

<table>
<thead>
<tr>
<th>Level</th>
<th>Requirements on the Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>It sets Short X-Locks</td>
</tr>
<tr>
<td>1</td>
<td>It sets Long X-Locks</td>
</tr>
<tr>
<td>2</td>
<td>It sets Long X-Locks &amp; Short S-Locks</td>
</tr>
<tr>
<td>3</td>
<td>It sets Long X-Locks &amp; Long S-Locks</td>
</tr>
</tbody>
</table>

Table 5.3. Implementational Definition of the 4 transactional levels.

Thus each transaction belongs to a given combination or class depending on what kind of locks it sets. It is supposed that all transactions belong to one or another of the 4 classes, but not necessarily the same.

Note that the requirements of each class are subsumed by those of the next one in order. So the term levels (of non-serializability) is appropriate. Further observe that the level 3 definition - saying that no lock is released before transaction-commit, is strictly stronger than the 2PhaseLocking criterion of Section 2.3.2 - saying that no lock is released before all locks are set.

Knowing that all transactions at least follow the requirements of level 0 - i.e. at least set short exclusive-locks, the effects from an operational point of view of a specific transaction belonging to a given level may be analyzed. The results are shown in Table 5.4.

The uncommitted-writes of other transactions mentioned for level 0 and 2, only exist for transactions which themselves belong to level 1, 2
Table 5.4. Operational Specification of the 4 transactional levels.

<table>
<thead>
<tr>
<th>Level</th>
<th>Effects of the Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>It does not Update Uncommitted-Writes or Uncommitted-Reads of other transactions</td>
</tr>
<tr>
<td>1</td>
<td>0 + It does not Commit its Uncommitted-Writes before Transaction-Commit</td>
</tr>
<tr>
<td>2</td>
<td>1 + It does not Retrieve Uncommitted-Writes of other transactions</td>
</tr>
<tr>
<td>3</td>
<td>2 + It does not Commit its Uncommitted-Reads before Transaction-Commit</td>
</tr>
</tbody>
</table>

or 3. Remember that transactions on level 0 only set short exclusive-locks. Thus they have no uncommitted-writes (see Table 5.1).

The uncommitted-reads of other transactions mentioned for level 0, only exist for transactions which themselves belong to level 3. Remember that transactions on level 0 and 1 set no shared-locks, while transactions on level 2 only set short shared-locks. Thus in both cases they have no uncommitted-reads (see Table 5.2).

Note also that the effects - in terms of consistency preservation (and reliability assurance) - on each level are subsumed by those on the next one in order. So it is appropriate to talk about different degrees of consistency (as in [Gray76]).

First, let us look at which recovery and concurrency control problems are possible at the different levels. This is shown in Tables 5.5 and 5.6. The listed problems may occur even if just one transaction belongs to the class corresponding to the indicated level, and all the other transactions belong to the class corresponding to level 3.

Table 5.5 covers the problems that may occur when each transaction either only retrieves or updates an item (and not both retrieves and updates an item). These are the two recovery oriented problems corresponding to cases iii) and iv) in Section 2.2.5, and the concurrency control oriented problem corresponding to case ii) in the same section. Thus the R-subscript on lost update here indicates recovery orientation.

Table 5.6 covers both the concurrency control oriented problems. As mentioned above, the one corresponding to case ii) in Section 2.2.5 requires only a retrieval or an update of an item of each transaction. However the other corresponding to case i) in the same section requires both a retrieval and an update of an item of at least one transaction. This combined reading and writing may be approached in several alternative ways, and some options will be discussed below. Thus the C-subscript on lost update now indicates concurrency control orientation.

The distinction between only 1 lock per item and both 1 and 2 locks per item is based on the not-read-before-write case of Section 2.2.4. However both tables have the same contents also for the read-before-write case of the same section. Observe that for the model used here, level 3 is free of all recovery and concurrency control problems.
Table 5.5. Problems possible by just one transaction executing at the indicated level as a maximum (with only 1 lock per item).

<table>
<thead>
<tr>
<th>Level</th>
<th>Recovery &amp; Concurrency Control Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Lost Update + Invalid Retrieval + Inconsistent Retrievals</td>
</tr>
<tr>
<td>1</td>
<td>Invalid Retrieval + Inconsistent Retrievals</td>
</tr>
<tr>
<td>2</td>
<td>Inconsistent Retrievals</td>
</tr>
<tr>
<td>3</td>
<td>——</td>
</tr>
</tbody>
</table>
The invalid retrieval problem which is possible at level 1 (and 0), is illustrated in Fig. 5.6. This is basically the same example as the first one of the two cases used in Fig. 2.55.

Fig. 5.6. Retrieval R₁(y) gets invalid at abort A₀.
Transaction T₁ executes at level 0 or 1.
and T₂ executes at 3 (eventually 2, 1 or 0).

The inconsistent retrieval problem which is possible at level 2 (and 1 or 0), is illustrated in Fig. 5.7. This is a variation of the example used in Fig. 2.49 in Section 2.3.2.

Fig. 5.7. Retrieval R₁(y) gets inconsistent
together with and at retrieval R₁(z).
Transaction T₁ executes at level 0, 1 or 2.
and T₂ executes at 3 (eventually 2, 1 or 0).
Table 5.6 needs some further explanation.

An item which is both to be read and written, may be locked in several different ways. One option is to request the X-lock needed during the update even prior to the retrieval (and release it after the update). Still the X-lock may either be short or long. This alternative is covered by the One-Lock column. Another option is to have the item only S-locked (eventually) during the retrieval and X-locked only during the update. Thus on levels 0 and 1 the item will not be locked at all during the retrieval. On level 2 the retrieval will be protected by a short S-lock. While on level 3 the retrieval will be protected by a long S-lock. Hence on level 2 there are two different possibilities. The short S-lock may be released immediately after the retrieval. This alternative is covered by the ReLock column. The short S-lock may also be released only after the X-lock is granted. This alternative is covered by the Upgrade Lock column.

Specific examples supporting all the different entries in the table are easy to construct. (Refer back to Fig. 2.14 in Section 2.2.5 for the lost update problem with concurrency control orientation. Let transaction T₁ with or without retrieval R₁(x) - always execute at level 3, and let transaction T₂ varyingly execute at level 0, 1, 2 or 3). But the examples have been excluded from the text because of the high number of combinations. We will cover a generalized set of concurrency control oriented problems in Section 9.3.

All the three alternatives are possible interpretations. The ReLock alternative corresponds to the widest classes of schedules. It will be the basic reference when we return to this material in Section 9.4.1. (See also Section 9.1.2). The Upgrade Lock alternative corresponds to a 2PhaseLocking criterion per single item. (Notice that the 2PhaseLocking criterion of Section 2.3.2 is per multiple item).

Tables 5.5 and 5.6 show that the class-division corresponding to levels 0, 1, 2 and 3 is fine-grained on recovery but coarse-grained on concurrency control. (This applies to both the ReLock and One-Lock alternatives). Thus the class-division of [Gray76] seems to be more recovery oriented and less concurrency control oriented.

Second, let us look at which concurrency control and recovery properties (see Section 2.3.2) are guaranteed at the different levels. This is shown in Table 5.7. Compare this with Table 2.2 in Section 2.3.2.

Table 5.7. Properties guaranteed by all transactions executing at the indicated level as a minimum.

<table>
<thead>
<tr>
<th>Level</th>
<th>Concurrency Control &amp; Recovery Properties</th>
<th>⪰: Cᵣ ⊇ VSR &amp; Cᵢ ⊇ RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Recoverability of Writes with regard to Writes</td>
<td>⪰: Cᵣ ⊇ VSR &amp; Cᵢ ⊇ RC</td>
</tr>
<tr>
<td>1</td>
<td>1 + Recoverability of Writes with regard to Reads</td>
<td>⪰: Cᵣ ⊇ VSR &amp; Cᵢ ⊆ ST</td>
</tr>
<tr>
<td>2</td>
<td>2 + Repeatability of Reads</td>
<td>⪰: Cᵣ ⊆ 2PL &amp; Cᵢ ⊆ ST</td>
</tr>
</tbody>
</table>

Cᵣ indicates: The class of schedules corresponding to level i
The mentioned class-relationships are again only rough indications of how the classes are related. (A $\supset$ is supposed to mean more inclusive than a $\supset$, and a $\supset\supset$ is supposed to mean much more inclusive than a $\supset$). Here we are only concerned with the basic problems introduced in Sections 2.2.5 and 2.3.2, and the mentioned class-relationships are thus not meant to have general validity. A formally correct and detailed analysis of classes $C_0$ - $C_3$ will be included in Section 9.4.1. (Note that class $C_i$ comprises all schedules in which each transaction executes at at least level $i$).

The relations between the classes analyzed in Section 2.3.2 and this section are:

- $C_C = C_3$ \hspace{2cm} (Eq. 5.1)
- $C_B = C_2$ \hspace{2cm} (Eq. 5.2)
- $[C_A \neq C_1] \land [C_A \neq C_0]$ \hspace{2cm} (Eq. 5.3)

Hence the entries for levels C and B respectively in Tables 5.5 and 5.6 will be the same as for levels 3 and 2. Further the entries for level A will be the same as for level 0 in Table 5.5 and the same as for level 2 in Table 5.6.

Going from LockUntilEnd to 2PhaseLocking (see Section 2.3.2) induces no loss with respect to any of the given concurrency control properties in either Table 2.2 or Table 5.7. (For level 3 [$=C$] it means that the long X- and S-locks only have to observe the 2PL-restriction among themselves. For levels 2 [$=B$] and 1 it means that even only the long X-locks have to observe the 2PL-restriction among themselves). Vice versa going from 2PhaseLocking to LockUntilEnd induces no gain with respect to any of the treated concurrency control problems in Table 5.6. However this is definitely not true for the recovery properties or recovery problems, as discussed in Section 2.3.2.

The reason for the discrepancy between the two cases corresponding to Table 5.7 and Table 2.2 is the use of different sets of lock-combinations. Fig. 5.8 clarifies how the two sets are related. It is a collocation of Fig. 5.4 in this section and Fig. 2.53 in Section 2.3.2.

The levels and corresponding labels that we have investigated, are those of [Gray76]. [Date83] operates with a slightly different set. A comparison of the levels and labels used in the two sources is given in Table 5.8.

The main difference between the two approaches is that a single action (write or read) in [Gray76] corresponds to two different actions in [Date83]. Write translates into find + update, while read translates into find + get. Further observe that [Date83] also considers a problem not covered by [Gray76] - namely the phantom items issue (see Section 2.3.2). This is the reason for the extra fifth level.
Fig. 5.8. Correspondence between Lock-combinations analysed in Section 5.3 [0, 1, 2, 3] and in Section 2.3.2 [A, B, C].

Table 5.8. A comparison of the Levels and Labels of two given sources.

<table>
<thead>
<tr>
<th>[Gray 76]</th>
<th>[Date 83]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Level 0 of [Gray 76] corresponds to:
An Item is Tested for Exclusive-Locking, but only "immediately" Marked with an X-Lock

Level 2 of [Date 83] corresponds to:
1 + An Item is Tested for Shared-Locking, but not Marked with an S-Lock at all

Level 5 of [Date 83] corresponds to:
4 + The Access-Path of an Item is Locked instead of the Item itself
Part IV
6 Skeleton-Databases

This chapter introduces our new type of distributed database, the skeleton-database, where some integrity constraints are lacking. This opens the way for the definition of alternative and additional correctness criteria instead of and on top of the preservation of integrity constraints.
6.1 **Traditional Distributed Databases**

As described in Section 3.1.3, most distributed databases have aspects both of localization and duplication. Localization means dividing one initially centralized database into several parts placed at different sites. Duplication means making several copies of one or more parts of an initially centralized or decentralized database and spreading these over several sites.

We will in this section concentrate on the duplication aspect. The motive for duplication is mainly increased availability, as given in Section 3.1.3.

Here, let us imagine a banking organization having three complete copies of their accounting information stored on three different computers in three separate branches. The general picture will be as in Fig. 6.1. Security reasons have led to a three-fold totally replicated database.

![Diagram](image)

**Fig. 6.1.** Existing interrelations in a Traditional Distributed Database.

The global database consists of a set of completely identical local databases. The three subunits are interrelated by equality, integrity constraints. They are supposed to be identical all three, item by item.

Within each local database there may of course be any other integrity constraints relating the local items somehow.
6.2 Alternative Distributed Database

Now, let us instead imagine a department store chain having three separate branches with completely identical selections of articles. However, the current prevailing prices, available qualities and remaining quantities of articles may vary among the department stores. Their article information will have to exist in three non-identical versions and may have to be stored on three different computers. The general picture will be as in Fig. 6.2. Diversity reasons have led to a three-fold non-replicated database. Thus we are now concentrating on the localization effect.

![Diagram](Image)

**Global Database**

**Collection of Non-Interrelated Local Databases**

Fig. 6.2. Lacking interrelations in the Alternative Distributed Database.

The global database consists of a set of local databases not being complete copies of each other. The three subunits are not interrelated by equality integrity constraints. They are not supposed to be identical, item by item.

But within each local database there may of course be any other integrity constraints relating the local items somehow.

Note that we anticipate that the local databases of each department store have entries for exactly the same set of articles. A relaxation of this restriction is included in Section 9.6.2. Even without this requirement the local databases do not merely constitute the results from several horizontal fragmentation operations on a global database, see Section 3.1.3 (anticipating a relational data model). A clarification of the distinction is given in Section 9.1.2. Hence this alternative database approach represents a new concept.

To get an immediate preview of typical examples of such alternative distributed databases, take a look at Section 9.5.
6.3 Consequences of Alternative Database Approach

Still there are some similarities between the two types of distributed databases illustrated in Figs. 6.1 and 6.2. In Fig. 6.1 we have several copies of the same database-occurrence, while in Fig. 6.2 we have several occurrences of the same database-type. Thus in the first case the local databases have the same content, while in the second case the local databases only have the same schema. Hence in the traditional case there will be equality constraints between database-values, while in the alternative case there will only be equality constraints between database-structures.

Generally, in a traditional database there may be any kind of integrity constraints on any kind of levels - e.g. between the separate local databases in a global database or between the separate items in a local database. As discussed in Section 4.3, the integrity constraints themselves are usually considered to be unknown due to the problems of representation and manipulation. This induces the needs and theory of serializability, again see Section 4.3.

What we have done, is to go from the more general case to a more special case. We are actually specifying an alternative database type where some integrity constraints - i.e. those between the separate local databases in a global database - are completely lacking. Hence we have complete knowledge about a well-defined part of the integrity constraints. As the non-existence of integrity constraints between clearly identifiable database parts is easy to represent and manipulate, this permits at least two non-exclusive possibilities:

i) Knowledge about some integrity constraints may be utilized by the system scheduler to achieve more efficient transaction scheduling. This allows non-serializable transactions without corrupting the database consistency. Hence we bypass the theory of serializability of Section 4.3.

ii) Lack of some integrity constraints opens the way for the definition of alternative and additional correctness criteria instead of and on top of the preservation of integrity constraints.

In Chapter 9 we will only exploit possibility ii).

We choose to call a global database of this type a skeleton-database. It contains parts looking the same without being the same.
7 WANDER-TRANSACTIONS

This chapter introduces our totally new type of distributed transaction, the wander-transaction, where some overall semantics information is existing. This leads the way to the specification of concrete correctness criteria instead of and on top of the preservation of integrity constraints.
7.1 **Traditional Distributed Transactions**

As in Section 6.1, we will in this section concentrate on the **duplication** aspect of distributed databases.

Again let us look at the **banking organization** having three complete copies of their accounting information stored on three different computers in three separate branches. Such a three-fold totally replicated database was shown in Fig. 6.1 in Section 6.1.

Here, let us then imagine a simple, but typical, transaction **depositing** or **withdrawing** some **money** into/from a specific account. This is illustrated in Fig. 7.1.

![Fig. 7.1. Actions of a Traditional Distributed Transaction.]

The transaction senses an old value by retrieving **one** copy of a specified account, changes this old value into a new value by adding or subtracting a given amount, and sets the new value by updating **all** existing copies of the account.

For more composite transactions (e.g. transferring some money between two specific accounts) the picture will be more complex. Generally a transaction will initially read **few** database items, then compute **some** new database values, and finally write **many** database items.
7.2 Alternative Distributed Transaction

Again let us instead look at the department store chain having three separate branches with completely identical selections of articles, and with their article information existing in three non-identical versions stored on three different computers. Such a three-fold non-replicated database was shown in Fig. 6.2 in Section 6.2. Thus we are again concentrating on the localization effect of distributed databases, as in Section 6.2.

Now, let us then imagine a simple, but interesting, operation buying the cheapest version of a specific article. This is illustrated in Fig. 7.2.

![Fig. 7.2. Actions of the Alternative Distributed Transaction.]

The operation checks the current offers by retrieving all existing versions of a specified article, finds one cheapest offer by comparing the prices and consulting the quantities, and acquires the best offer with at least one instance left by updating this version of the article.

For more composite operations (i.e. buying the optimal set of versions of several specific articles - given any combined predicate on prices, qualities, and quantities) the picture will be more complex. Generally an operation will initially read many database items, then compute few new database values, and finally write some database items.

To check an offer means to retrieve its price-, quality-, quantity- and purchase-attributes. A purchase-list reflects the sequence of reservations already made. To acquire an offer means to decrement the quantity (≥ 1) by one and add the buyer-identity to the purchase-list. The purchase-attribute will not be included in the illustrations - to save space. However it will produce a delivery-list for the department store, so the sequence of buyer-identities is important.

This alternative transaction approach represents a totally new concept. We will give a further clarification of the concept in Sections 9.1.2 and 9.2.1.

To get an immediate preview of typical examples of such alternative distributed transactions, again take a look at Section 9.5.
7.3 Consequences of Alternative Transaction Approach

Generally, a traditional transaction reads some database items and writes some database items. The set of items read and the set of items written are themselves related somehow, and they are used in implementing some function. As indicated in Section 4.3, both the relation between the read-set and write-set of a transaction (high-level syntax information) and the functional objective of a transaction (overall semantics information) are usually considered to be unknown - due to the dynamic behaviour of a single transaction and the dynamically changing set of transactions. This brings about the non-existence of polynomial time solutions to some scheduling problems, again see Section 4.3.

What we have done, is again to go from the more general case to a more special case. We are actually specifying an alternative transaction type where the relation between the read-set and write-set of the transactions is known, and the functional objective of the transactions is fixed. Hence we have both some high-level information about the syntax of the transactions and some overall information about the semantics of the transactions. Separately concentrating on a specific type of transaction thus permits at least two non-exclusive possibilities:

i) Knowledge about some high-level syntax information may be utilized in the system scheduling to achieve more efficient transaction scheduling. The read-before-write case induces polynomial time solutions to more scheduling problems. Hence we exploit the theory of serializability of Section 4.3.

ii) Existence of some overall semantics information leads the way to the specification of concrete correctness criteria instead of and on top of the preservation of integrity constraints.

In Chapter 9 we will exploit both possibility ii) and i).

We choose to call a transaction of this type a wander-transaction. It traverses a skeleton-database trying to seize an optimal set (complying with any combined conditions on the attribute-values) of specific database items.
8 DISCUSSION OF SKELETON-DATABASES AND WANDER-TRANSACTIONS

In this chapter we carry out a general and overall analysis of the use of the new types of distributed database and distributed transaction. The new systems tolerate breaks both with the one-after-the-other atomicity, the all-or-nothing atomicity and the all-or-none atomicity/unity neither sacrificing the database correctness nor the transaction correctness.
8.1 Possibilities of Alternative Systems

An elaboration of the options resulting from defining and using specialized databases and/or transactions will be based on the available knowledge about the databases and/or transactions.

8.1.1 Information

The knowledge about a database and its transactions that may be accessible and thus possibly used by the system scheduler, is usually considered to fall into three groups as indicated in Table 8.1. See also Sections 2.2.7 and 4.3.

Table 8.1. Different types of knowledge possibly accessible to the scheduler.

<table>
<thead>
<tr>
<th>Possible Knowledge of the Scheduler</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax : Which Items are Read and Written</td>
<td>R_a(x), W_b(y)</td>
</tr>
<tr>
<td>Semantics : How are New Values Computed from Old Values</td>
<td>y := x</td>
</tr>
<tr>
<td>Integrity Constraints : What Conditions are the Items' Values to Fulfil</td>
<td>x = y</td>
</tr>
</tbody>
</table>

According to this traditional division, the first two categories, syntax and semantics, deal with the transactions, while the last one, integrity constraints, deals with the database.

We feel that it is natural to employ a little more fine-grained splitting. In this alternative division we also look upon some knowledge about the database as semantics; e.g. information stating that the items are arranged in trees, graphs or tables. Thus syntax information still only concerns the transactions. It deals with access information; i.e. which items are used - selected, and how are these items used - read or written. But semantics information now both concerns the transactions and the database. It deals with functional information - i.e. which mappings do the actions of a transaction implement logically, or structural information - i.e. how are the items of a database organised physically. And integrity constraints information again only concerns the database. It deals with relational information; i.e. how are the items of a database connected logically.

These three categories may be combined in different ways. The effects of the use of some such combinations are shown in Table 8.2.

Those combinations with "Investigated" in the actuality column have earlier been analyzed in [Kung83]. and those tagged "Interesting" are now to be investigated by us. The optimal scheduler column gives the widest class of schedules (see Section 4.3) that a scheduler with the given knowledge possibly could leave untouched without risking corrupting the correctness of database-behaviour. (Further see [Papa83] and [Yann82a] for in-depth investigations of the possibilities and limitations of strategies restricted to locking only).
Table 8.2. Applicability and results of different combined types of knowledge.

<table>
<thead>
<tr>
<th>Given Knowledge of a Scheduler</th>
<th>Actuality</th>
<th>Optimal Scheduler</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; Nothing &gt;</td>
<td>Investigated</td>
<td>S</td>
</tr>
<tr>
<td>Syntax</td>
<td></td>
<td>FSR</td>
</tr>
<tr>
<td>Syntax + Semantics</td>
<td></td>
<td>WSR</td>
</tr>
<tr>
<td>Syntax + Integrity Constraints</td>
<td>Interesting</td>
<td>ASR</td>
</tr>
<tr>
<td>Syntax + Semantics + Integrity Constraints</td>
<td>Investigated</td>
<td>ASR*</td>
</tr>
<tr>
<td>Dynamic Syntax Indication</td>
<td>Investigated</td>
<td>2PL</td>
</tr>
</tbody>
</table>

Dynamic Syntax means: To be known during execution, and not in advance

WSR (weakly serializable) is a "new" class from [Kung83], while ASR (alternative-to-serializable) and ASR* (extended alternative-to-serializable) are the two new classes designated, named and analyzed by us. Two special versions of ASR and ASR* respectively are of particular interest to us:

i) ASR with the lack of some integrity constraints applies to skeleton-databases.

ii) ASR* with the lack of some integrity constraints and with the existence of some overall semantics information applies to skeleton-databases plus wander-transactions.
8.1.2 **Class WSR**

Let us just look at two examples illustrating the use of semantics information.

First, we will focus on the use of semantics information about the transactions.

Usually the only actions of a transaction taken into account are read and write. Table 8.2 in the previous section shows that with even static syntax information, these conditions correspond to FSR as the optimally scheduled class. But often some actions make up some simple high-level actions like increment (corresponding to a read, an addition and a write) or decrement (corresponding to a read, a subtraction and a write); i.e. composite actions with known functions. Taking such extra semantics knowledge into account - i.e. considering that static syntax information plus semantics information about the transactions are available, different Non-FSR schedulers would still preserve consistency. (See also Section 2.2.7).

These ideas have been applied by [Reut82] to locking on heavily-write-accessed items. The essence is that schedules being non-serializable on the read/write-level, may be serializable on the read/write/increment/decrement-level. A compatibility matrix with read, write, increment and decrement (and not only read and write as usual) will actually make two increments, one increment plus one decrement and two decrements all possible for parallel execution.

Second, we will focus on the use of semantics information about the database.

Usually the items of a database are considered to be unstructured. Table 8.2 shows that with only dynamic syntax information, these conditions correspond to 2PL as the optimally scheduled class. But often the items of a database are actually structured; e.g. organised physically as trees or graphs. Taking such extra semantics knowledge into account - i.e. exploiting that dynamic syntax information plus semantics information about the database are available, different Non-2PL schedulers will still guarantee serializability. (See also Section 2.2.7).

These ideas have been applied by [Silb80] and [Kede83] to locking in hierarchical structures. They treat respectively exclusive-locks and exclusive-locks plus shared-locks. [Kede81] also investigates the limits of these types of systems; i.e. using semantics information about the database. Observe that such physical tree/graph locking is different from the logical tree/graph locking in multigranularity systems, see Section 2.3.2.
8.1.3 **CLASS ASR**

A closer look at ASR (and its special version) will be based on Fig. 8.1.

![Diagram showing relationships between WSR, ASR, and FSR]

**Fig. 8.1.** Different ways of using the fact that some integrity constraints are known.

The main point is that ASR (and its dynamically defined special version) is a wider class than FSR. Thus ASR allows different types of non-serializable transactions still yielding correct database behaviour. Effectively, non-serializability means that fewer reads and writes have to wait. Hence more parallelism is achieved.

Paraphrased: ASR with skeleton-databases breaks with the usual one-after-the-other atomicity of transactions (see Sections 2.2.2 and 3.2.1). We are basically introducing a new "transaction" concept and approach. Effectively, true parallel behaviour may now still give sensible results.
8.1.4 **Class ASR**

We will also take a closer look at ASR* (or rather its special version).

Operating traditional transactions on a traditional database involves a certain kind of rigid total treatment. If and when a database item is logically non-reachable to a transaction (due to interference) or physically non-reachable to a transaction (due to failures, or because of decision making), that transaction is always blocked or aborted. There is never an alternative – due to correctness considerations.

On the contrary, operating wander-transactions on a skeleton-database allows rather a certain degree of non-rigid optimal treatment. According to the semantics of wander-transactions and skeleton-databases, they do not always have to go for the one-and-only best solution. Instead they might go for the best available solution; i.e. seize and use the database items logically and physically reachable (restricted somehow) at any time.

Paraphrased; ASR* with skeleton-databases and wander-transactions breaks with as well the usual one-after-the-other atomicity of transactions (again see Sections 2.2.2 and 3.2.1) as the usual all-or-nothing atomicity and all-or-none atomicity/unity of transactions (see Sections 2.2.2 and 3.2.2). We are basically refining our new "transaction" concept and approach. Effectively, both true parallel and true partial behaviour may now still give sensible results.
8.1.5 Concept Relations and Development

A collocation of already existing ideas and our new ideas about the notions of serializability and non-serializability (using integrity constraints information and semantics information) is given in Fig. 8.2.

![Diagram](image)

Fig. 8.2. The conceptual context of ASR*: i.e. Alternative-to-Serializability*.

Let us clarify how these ideas and notions are related, and how they have developed.

- 2PL Serializability:

  The initial ideas are from [Eswa76].

* 2PL Serializability = 2PL Non-Serializability:

[Gray76] effectively analyzed what might be achieved through different combinations of locks. The investigated options were X- and S-locks, and both types could have no, short and long variants. The resulting classes were C0, C1, C2 and C3. See Chapter 5. These classes were both non-serializability and non-recoverability oriented, but the bias was towards non-recoverability.

* 2PL Non-Serializability ⇒ Alternative-to-Serializability*:

This thesis effectively investigates what ought to be achieved for different specifications. The treated cases are read-before-write and not-read-before-write. The resulting classes are A_RSR and A_NR_SR. See Chapter 9 (and Chapter 10).
These classes are both non-serializability and non-recoverability oriented (and even non-atomic-commitment oriented), but the bias is towards non-serializability.

* 2PL Serializability → General Serializability:

Different extensions from one to several classes of serializable schedules were presented in [Bern79b], [Papa79], [Bern87b] and [Papa86]. [Bern79b] and [Papa79] described a lot of interesting classes for the single-action-multi-item model. Among these were FSR, SSR, CSR, WPL, 2PL, CGS, P3P and SDD. See Chapter 4. [Bern87b] and [Papa86] described a few actual classes for the multi-action-single-item model. Among these were FSR, VSR_0, VSR_N and CSR. See Chapter 2 and the Appendix.

* General Serializability → Alternative-to-Serializability*:

A development from classes of serializable schedules to classes of non-serializable schedules is described in this thesis. The derivation of A^SR and A^M^SR is CSR-based. See Chapter 9 (and Chapter 10). A discussion of the possibilities of basing the alternative correctness criteria on other classes than CSR (i.e. specifically FSR) is also included. Again see Chapter 9 (and Chapter 10).

- Alternative-to-Serializability*:

[Lamp76], [Kung80] and [Lehm81] all discuss systems of our type with non-serializable transactions that still yield correct database-behaviour. The difference from our approach is that all three base their system scheduling on both the knowledge about (and not only the lack of) integrity constraints and the existence of detailed (and not only overall) semantics information. [Lamp76] philosophizes over a banking-database system - based on integrity constraints information about the database and semantics information about the transactions. [Kung80] and [Lehm81] respectively describe binary-tree and B-tree systems - both based on integrity constraints information about the database and semantics information about the transactions plus the database. In one respect [Lamp76] is closer to our approach than [Kung80] and [Lehm81]; i.e. we base the investigations of these types of systems on integrity constraints information about the database and semantics information about the transactions - without semantics information about the database.

An interesting coupling of some other existing ideas is described in [Kort88b]. It is oriented towards long-duration transactions in CAD or CAM-like applications. The associated predicate correct notion is a generalization of the "serializability" notion through the combination of three already known concepts. All these are themselves extensions of the serializability concept.
- Multiversion Serializability:

This concerns a coupling of concurrency control and recovery through the maintenance of several non-identical versions of each item, see Section 2.3.2 (and Section 2.2.8). A general theory was given in [Bern83a] and [Papa84], as mentioned in the same section. A schedule which is non-serializable with only one version per item, may still be serializable with several versions per item. This depends on semantics information about the database (unlike in our approach). See also the treatment of such non-serializability in Section 9.1.2.

- Partial Order Serializability:

The name refers to the non-total ordering of actions which is possible in a transaction, see Section 2.2.4. However the essence deals with a possible hierarchical organization of transactions. Such nested transactions were discussed for example in [Gray81a] and [Lync83]. [Banc85] initiated one of the developments of concurrency control policies and mechanisms for these systems. [Beer86] continued this work, and it is this source which [Kort88b] builds on. Nested transactions may lead to schedules which are non-serializable on a lower level but still serializable on a higher level. This depends on semantics information about the transactions (like in our approach). However an existence of detailed information and not only overall information is required (unlike in our approach).

- Predicatewise Serializability:

This concerns databases where the integrity constraints may be stated as predicates in conjunctive normal form. [Banc85] again initiated one of the developments of concurrency control policies and mechanisms for these systems. [Kort88a] continued this work, and it is this source which [Kort88b] builds on. Such predicated integrity constraints may lead to a dissection of the database (items) into independent (i.e. non-related) parts. Any schedule may be non-serializable between parts as long as it is serializable within each part. This depends on integrity constraints information about the database (like in our approach). However knowledge about them and not only their lack is required (unlike in our approach). See also the treatment of fragmentwise serializability in Section 9.6.2.

The resulting class is VSR₀-based, but a CSR-based variant is also included.
8.2 Effects of Alternative Database and Transaction Types

The examination of the consequences of defining and using skeleton-databases and wander-transactions will be divided into an overall treatment of some generic effects and a detailed treatment of some specific effects.

8.2.1 Generic Consequences

Let us review the contents of Section 3.1.3 with respect to which of the usual distributed database system objectives that may be eased for wander-transactions in a skeleton-database.

- Implement atomic transactions with regard to concurrency control and recovery.

&

Accept local site autonomy with regard to user- (and system-) processes.

These two objectives do not have to be fully observed in our alternative systems. For the first aspect this is true directly, while for the second aspect it is only true indirectly.

As we saw in Sections 8.1.3 and 8.1.4, we may allow as well true parallel behaviour among global transactions (corresponding to breaks with the one-after-the-other atomicity) as true partial behaviour within local databases (corresponding to breaks with the all-or-nothing atomicity) and true partial behaviour between local databases (corresponding to breaks with the all-or-none atomicity/unity).

These aspects will be further commented upon in the next section, and they will be covered in depth in Chapter 9.

- Optimize global access with regard to the use of data-items from multiple sites.

&

Assure global data transparency with regard to (data fragmentation, data replication and) data location between sites.

Even these two objectives do not have to be fully complied with in our alternative systems — eventually.

We will return to such aspects in Section 10.2.
- Allow data fragmentation and data replication with regard to user- (and system-)data.

Per definition there is "horizontal" data fragmentation, but no data replication in a skeleton-database. Hence all the extra issues and problems corresponding to the c) subcases of Section 3.1.7 do not need any attention.

Thus, with reference to Section 3.3.2, the now non-existing need of a common failure/non-failure view of physical item "copies" makes it possible for example to allow updates in multiple partitions, and not only in a single partition. However, with reference to Section 3.3.1, the now non-existing need of a common physical control-"copy" or fixed separate control-point for item "copies" will of course be replaced by another synchronization need. Which need, that is again to be the basic theme of Chapter 9.
8.2.2 Specific Consequences

First, let us look at the one-after-the-other atomicity.

As indicated in Fig. 8.1 in Section 8.1.3, the lack of some integrity constraints in a skeleton-database leads to certain degrees-of-freedom in synchronizing accesses to such databases. The existence of some overall semantics information about a wander-transaction then fixes some specific points in this solution span for synchronization of such transactions. Chapter 9 actually pursues what the generic solution span encompasses, and how the specific solution points are clarified.

As an example we will comment on the eventual need for 2PhaseLock synchronization in such systems. This notion was introduced in Section 2.2.7, and the mechanism is described in full in Sections 2.3.2 and 3.2.1. It corresponds to the optimal scheduler for dynamic syntax information only as shown in Table 8.2 in Section 8.1.1.

Is 2PhaseLocking necessary for executing wander-transactions on a skeleton-database? The answer is negative. It is quite adequate to always hold the best offer currently found and constantly free already obsolete offers. Holding corresponds to locking and freeing to unlocking. Carrying out this principle recursively means that eventually an article is locked after another is unlocked. Hence 2PhaseLocking is violated.

Second, let us look at the all-or-nothing atomicity and the all-or-none atomicity/unity.

A skeleton-database comprises by its very definition a set of centralized databases with a high degree of local site autonomy. Local failures within the separate databases and global failures among the separate databases might come and go. The level of cooperation between the separate local databases and the level of authorization of the separate global users might also differ. These variations in access-options and access-rights imply that it will not be possible for all users to acquire everything in such systems.

Further a wander-transaction might by its very definition be more interested in usable, time-saving solutions than exact, time-spending solutions. The semantics of wander-transactions and skeleton-databases indicates a possible trade-off between the time-delay induced by the (logical and) physical blocking/abortion from (concurrency control and) recovery - and a degree of goal-fulfilment with respect to the stated correctness criteria. These reflections about optimal vs. total results imply that it will not even be necessary for all users to acquire everything in such systems.

As an example we will also comment on the eventual need for 2PhaseWrite treatment in these systems. This notion was introduced in Section 3.1.6, and the mechanism is effectively described in Sections 2.4.2 and 3.2.2. Normally it is a definite minimum within cooperating autonomous sites.
Is 2PhaseWriting necessary for terminating wander-transactions on a skeleton-database? Again the answer is negative. It is quite adequate not always to require either all or nothing when a specific set of articles is sought. It is quite possible sometimes only to require either all or nothing within - larger or smaller - separate groups of articles. The set of articles would have to have an implicit hierarchy built upon them, and each transaction might experience a step-wise termination. Compare this with the nested transactions where the transactions themselves are hierarchically organised, see Section 8.1.5.

As an example we will even comment on the eventual need for 2PhaseCommit treatment in these systems. This notion was introduced in Section 3.1.4, and the mechanism is described in full in Section 3.2.2. Normally it is a definite minimum among cooperating autonomous sites.

Is 2PhaseCommitment necessary for controlling wander-transactions on a skeleton-database? Once more the answer is negative. It is quite adequate not always to require either all or none when a specific set of sites is involved. It is quite possible sometimes only to require either all or none among the - at a fixed point in time - available sites. Each transaction might experience a step-wise control.
8.3 Cofunctioning of Alternative and Traditional Systems

With reference to Table 8.2 in Section 8.1.1, let us name the correctness criteria corresponding to the FSR, ASR and ASR* classes of schedules respectively SER, ASER and ASER*.

With reference to Fig. 6.2 in Section 6.2, we have until now needed and used:

ASER* and SER criteria each on one separate level

Some plausible extensions are:

1) ASER* and SER criteria on one common level

2) SER criteria on several levels

3) ASER* criteria on several levels

Actually cases 1) and 2) are logical, and Fig. 8.3 illustrates an example including both cases.

The different levels might for example correspond to the following points of view:

- Global Buyer:

  Wants to acquire the optimal set of versions of several specific articles among all stores.

  \(\rightarrow\) Global ASER*; i.e. High-Level, Alternative

- Local Owner:

  Wants to change the prevailing prices of all articles in a store.

  \(\rightarrow\) Local SER; i.e. Low-Level, Traditional

- Global Supplier:

  Wants to change the remaining quantities of an article in all stores.

  \(\rightarrow\) Global SER; i.e. High-Level, Traditional
However case 3) is not logical, as Fig. 8.4 illustrates.

Remember that according to our definition, ASER and ASER* imply equality constraints between database-structures - or matching schema. Thus several ASER* levels necessarily collapse into one single ASER* level.
Fig. 8.4. Non-existence of Alternative Criteria on several levels.
8.4 Evaluation of Alternative Database and Transaction Types

The highlights of the present ideas may be grouped into three:

- Novelty:

  Both skeleton-databases and wander-transactions as defined and used here are new concepts and indicate new approaches.

- Importance:

  There has been a recognition over the last few years of the usefulness in developing some special solutions to specialized problems, rather than finding more general solutions to generalized problems. The skeleton-database and wander-transaction concepts represent respectively specialized database and specialized transaction organisations.

- Applicability:

  With a personal computer available, the skeleton-database and wander-transaction approach allows us for example to do home-based electronic shopping in several separately located department stores. This really should be something for the future.
9 Theory for Skeleton-Databases and Wander-Transactions

In this chapter we carry out a specific and detailed analysis of the use of the new types of distributed database and distributed transaction. The new systems will correspond to two different non-serializability criteria, one included in the other, for the cases of not-read-before-write and read-before-write respectively.
9.1 **Options for Alternative-to-Serializability**

In this section we will outline and evaluate the solution span for a specification of correctness criteria for the class ASR* from Section 8.1.4.

9.1.1 **Mutual-Exclusion versus Serializability**

Here we shall describe two extreme points in a solution span for the above-mentioned class ASR*. We will base these on the two different models presented in Section 2.1.2; i.e. the operating systems model and the database systems model.

**Operating-systems criteria**

First, let us look at a common correctness criterion for the operating systems model. As mentioned in Section 2.1.1 the idea is to implement so-called atomic actions.

Eq. 2.6 in Section 2.2.4 defined conflicting operations as a binary relation (~) between actions. From this we need to specify some additional binary relations between transactions.

Initially we define three simple binary relations between transactions containing actions that are conflicting operations of the same kind and on the same item:

- \( WR_x(H) = \{(T_j, T_i) | W_j(x) <_h R_i(x)\} \) (Eq. 9.1)

- \( RW_x(H) = \{(T_j, T_i) | R_j(x) <_h W_i(x)\} \) (Eq. 9.2)

- \( WW_x(H) = \{(T_j, T_i) | W_j(x) <_h W_i(x)\} \) (Eq. 9.3)

Further we define three compound binary relations for conflicting operations of two and two different kinds but still on the same item:

- \( WR\cdot RW_x(H) = WR_x(H) \cup RW_x(H) \) (Eq. 9.4)

- \( WR\cdot WW_x(H) = WR_x(H) \cup WW_x(H) \) (Eq. 9.5)

- \( RW\cdot WW_x(H) = RW_x(H) \cup WW_x(H) \) (Eq. 9.6)

Finally we define one compound binary relation for conflicting operations of all three different kinds though still on the same item:

- \( WR\cdot RW\cdot WW_x(H) = WR_x(H) \cup RW_x(H) \cup WW_x(H) \) (Eq. 9.7)
Note that all the seven binary relations above are defined per single item. Thus we have such a set of binary relations for each separate item - \( x \in D \) - in the database.

Now we may specify an equivalence notion (like we did in Chapters 2 and 4) based on effects for each single item separately.

**x-Conflict equivalent schedules** is a Binary Relation \( \equiv_{xc} \):

\[
H \equiv_{xc} H' \iff
\begin{align*}
- & h = h' & \text{(Eq. 9.8)} \\
\text{and} \\
- & \forall (T_j, T_i) ((T_j, T_i) \in \text{WR-RW-WW}_x(H) \implies (T_j, T_i) \in \text{WR-RW-WW}_x(H')) & \text{(Eq. 9.9)}
\end{align*}
\]

Likewise we may specify two serializability notions (again as we did in Chapters 2 and 4) based on effects for each single item separately.

**x-Conflict serializable schedules** is a Set \( CMX_x \):

\[
H \in CMX_x \iff
\exists H_s \in S [H \equiv_{xc} H_s] & \text{(Eq. 9.10)}
\]

**Conflict mutual exclusive schedules** is a Set \( CMX \):

\[
H \in CMX \iff
\forall x \in D [\exists H_s \in S [H \equiv_{xc} H_s]] & \text{(Eq. 9.11)}
\]

Thus \( H \in CMX_x \) if and only if \( \text{WR-RW-WW}_x(H) \) is a partial order.

And \( H \in CMX \) if and only if all \( \text{WR-RW-WW}_x(H) - x \in D \) - are partial orders separately.

Observe that we are not requiring anything about the set of (hopefully) partial orders together.

To test class membership in a \( CMX_x - x \in D \) - for a given schedule \( H \), make a directed graph \( CSG_x(H) \) as:

\[
- V ( CSG_x(H) ) = \{ T_i | t_i \in h \} & \text{(Eq. 9.12)}
\]
A (CSG_x(H)) = (Eq. 9.13)

\{T_j \rightarrow T_i \mid W_j(x) \prec_h R_i(x)\} \quad (a)
\cup \{T_j \rightarrow T_i \mid R_j(x) \prec_h W_i(x)\} \quad (b)
\cup \{T_j \rightarrow T_i \mid W_j(x) \prec_h W_i(x)\} \quad (c)

Then H \in CMX_x if and only if CSG_x(H) is acyclic.

And H \in CMX if and only if all CSG_x(H) - x \in D - are acyclic separately.

Cases a), b) and c) are all illustrated in Fig 9.1. Observe that transitive arcs stemming from case c) between two and two transactions will be left out in the coming examples.

![Diagram](image)

Fig. 9.1. Resulting contributions to CSG(H) from different types of conflicts in H.

Note further that any topological sort of an acyclic CSG_x(H) corresponds to a serial schedule H_s x-conflict equivalent to H. Thus the given system schedule H may have several x-conflict serializations.

Observe that a conflict mutual exclusive schedule H may have different x-conflict serializations for each separate item x. There may not even be a consistent set of serialization order choices from the different sets of serialization orders for each separate item. In the worst case there may be as many as m = |D| - being the number of items in the database - mutually inconsistent serialization orders.
DATABASE-SYSTEMS CRITERIA

Second, let us look at a common correctness criterion for the database systems model. As mentioned in Section 2.1.1 the idea is to implement so-called atomic transactions.

Eqs. 9.1 to 9.7 defined a set of binary relations per single item. From these we need to specify some additional binary relations per multiple item.

Initially we define three simple binary relations between transactions containing actions that are conflicting operations of the same kind but possibly on different items:

\[- \text{WR}(H) = \{(T_j, T_i) | \exists x \in D \ [W_j(x) \prec_h R_i(x)]\} \]  \hspace{1cm} (Eq. 9.14)
\[\therefore \text{WR}(H) = \bigcup_{x \in D} \text{WR}_x(H)\]

\[- \text{RW}(H) = \{(T_j, T_i) | \exists y \in D \ [R_j(y) \prec_h W_i(y)]\} \]  \hspace{1cm} (Eq. 9.15)
\[\therefore \text{RW}(H) = \bigcup_{y \in D} \text{RW}_y(H)\]

\[- \text{WW}(H) = \{(T_j, T_i) | \exists z \in D \ [W_j(z) \prec_h W_i(z)]\} \]  \hspace{1cm} (Eq. 9.16)
\[\therefore \text{WW}(H) = \bigcup_{z \in D} \text{WW}_z(H)\]

Further we define three compound binary relations for conflicting operations of two and two different kinds and still possibly on different items:

\[- \text{WR-RW}(H) = \text{WR}(H) \cup \text{RW}(H) \]  \hspace{1cm} (Eq. 9.17)
\[\therefore \text{WR-RW}(H) = \bigcup_{x \in D} \text{WR-RW}_x(H)\]

\[- \text{WR-WW}(H) = \text{WR}(H) \cup \text{WW}(H) \]  \hspace{1cm} (Eq. 9.18)
\[\therefore \text{WR-WW}(H) = \bigcup_{y \in D} \text{WR-WW}_y(H)\]

\[- \text{RW-WW}(H) = \text{RW}(H) \cup \text{WW}(H) \]  \hspace{1cm} (Eq. 9.19)
\[\therefore \text{RW-WW}(H) = \bigcup_{z \in D} \text{RW-WW}_z(H)\]

Finally we define one compound binary relation for conflicting operations of all three different kinds also still possibly on different items:

\[- \text{WR-RW-WW}(H) = \text{WR}(H) \cup \text{RW}(H) \cup \text{WW}(H) \]  \hspace{1cm} (Eq. 9.20)
\[\therefore \text{WR-RW-WW}(H) = \bigcup_{x \in D} \text{WR-RW-WW}_x(H)\]
Note that all the seven binary relations above are defined per multiple item. Thus we have only one set of binary relations for all items together in the database.

Before we proceed, we introduce the following notational shorthand:

**Item projection** $T_i^x = T_i[x]$ of a transaction schedule $T_i$ with regard to $x$ is the Restriction of $T_i$ on $\{R_i(x) | R_i(x) \in t_i \} \cup \{W_i(x) | W_i(x) \in t_i \}$.

**Item projection** $H[x]$ of a system schedule $H$ with regard to $x$ is the Restriction of $H$ on $\cup_{t_i \in H} t_i^x$.

Now we may also specify an equivalence notion based on effects for all multiple items together.

**Conflict equivalent schedules** is a Binary Relation $\equiv_c$:

$$H \equiv_c H' \text{ iff }$$

- $h = h'$ \hspace{1cm} (Eq. 9.21)

and

- $\forall (T_j, T_i) [(T_j, T_i) \in \text{WR-RW-WW}(H) \iff (T_j, T_i) \in \text{WR-RW-WW}(H')]$ \hspace{1cm} (Eq. 9.22)

Hence with reference to $x$-conflict equivalence from Eqs. 9.8 to 9.9:

$$H \equiv_{xc} H' \text{ iff } H[x] \equiv_c H'[x]$$

Likewise we may also specify a serializability notion based on effects for all multiple items together.

**Conflict serializable schedules** is a Set CSR:

$$H \in \text{CSR} \text{ iff }$$

- $\exists H_s \in S [H \equiv_c H_s]$ \hspace{1cm} (Eq. 9.23)

Hence with reference to $x$-conflict serializability and conflict mutual exclusion from respectively Eq. 9.10 and Eq. 9.11:
\[ H \in \text{CMX}_x \iff H[x] \in \text{CSR} \]
\&
\[ H \in \text{CMX} \iff \forall x \in D \ [H[x] \in \text{CSR}] \]

Thus \( H \in \text{CSR} \) if and only if \( \text{WR-RW-WW}(H) \) is a partial order.

Or paraphrased; \( H \in \text{CSR} \) if and only if all \( \text{WR-RW-WW}_x(H) - x \in D \) are consistent partial orders.

Observe that we are now requiring something about the set of (hopefully) partial orders together.

To test class membership in \( \text{CSR} \) for a given schedule \( H \), make a directed graph \( \text{CSG}(H) \) as:

\[- V(\text{CSG}(H)) = \{T_i | t_i \in h\} \quad \text{Eq. 9.24}\]
\[- A(\text{CSG}(H)) = \]
\[
\{T_j \rightarrow T_i | \exists x \in D \ [W_j(x) \prec_n R_i(x)]\} \quad \text{(a)} \\
\cup \{T_j \rightarrow T_i | \exists y \in D \ [R_j(y) \prec_n W_i(y)]\} \quad \text{(b)} \\
\cup \{T_j \rightarrow T_i | \exists z \in D \ [W_j(z) \prec_n W_i(z)]\} \quad \text{(c)}
\]

Then \( H \in \text{CSR} \) if and only if \( \text{CSG}(H) \) is acyclic.

Or paraphrased; \( H \in \text{CSR} \) if and only if all \( \text{CSG}_x(H) - x \in D \) are acyclic separately and there is at least one consistent set of topological sorts of the separately acyclic \( \text{CSG}_x(H)s \).

Cases a), b) and c) are all illustrated in Fig 9.2. Again observe that transitive arcs stemming from case c) between two and two transactions will be left out in the coming examples.

![Fig. 9.2. Resulting contributions to CSG(H) from different types of conflicts in H.](image)

Note further that any topological sort of an acyclic \( \text{CSG}(H) \) corresponds to a serial schedule \( H_s \) conflict equivalent to \( H \). Thus the given system schedule \( H \) may have several conflict serializations.
Observe that while a conflict mutual exclusive schedule may have several mutually inconsistent serialization orders associated with each item separately, a conflict serializable schedule must have one consistent serialization order associated with all items together.

The class CSR used here is obviously "equal" to the one with the same name in Section 2.2.6. The specifications have been put in a slightly different form to facilitate other class specifications. Thus compare Eqs. 9.21 to 9.23 (and Eqs. 9.24 to 9.25) in this section with Eqs. 2.12 to 2.14 (and Eqs. 2.16 to 2.17) in Section 2.2.6. We have also left out the notion of a committed projection from the model presented in Section 2.2.4 (like with the model presented in Section 4.1). Actually in the following treatment we will basically work with the model in Chapter 2 adapted to the lack of the committed projection concept as in the model in Chapter 4. Specifically in this chapter and the next chapter we restrict the general versions of CSR and VSR_w to complete system schedules only (like VSR_0 and FSR). See also Section 4.3.
EXAMPLE OS

As a first explanatory example, we will start with the following schedule.

\[ H_1 = \]
\[ T_1: R_1(x)W_1(x) \quad R_1(y)W_1(y) \]
\[ T_2: R_2(x)W_2(x)R_2(y)W_2(y) \]

From Figs. 9.3a and 9.3b we see that:

\[ H_1 \leq x \leq T_1 \leq T_2 \]
\[ H_1 \leq y \leq T_2 \leq T_1 \]

Then we may conclude that

\[ H_1 \in CMX_x \]
\[ H_1 \in CMX_y \]

and hence that:

\[ H_1 \in CMX \]

---

**Fig. 9.3a. CSG_x[H_1].**

**Fig. 9.3b. CSG_y[H_1].**
But from a cyclic graph in Fig. 9.4 we have that

$H_1 \neq_c$ Any Permutation of $T_1$ & $T_2$,

and thus we must deduce that:

$H_1 \notin$ CSR

Fig. 9.4. CSG($H_1$).
EXHIBIT DS

As a second explanatory example, we will continue with the following schedule.

\[ H_2 = T_1: R_1(x)W_1(x) \quad \quad R_1(y)W_1(y) \]

\[ T_2: R_2(x)W_2(x) \quad \quad R_2(y)W_2(y) \]

From Figs. 9.5a and 9.5b we see that:

\[ H_2 =_{xc} T_1 \otimes T_2 \]

\[ H_2 =_{yc} T_1 \otimes T_2 \]

Then we may conclude that

\[ H_2 \in \text{CMX}_x \]

\[ H_2 \in \text{CMX}_y \]

and hence that:

\[ H_2 \in \text{CMX} \]

---

Fig. 9.5a. CSG\(_x\)(H\(_2\)).

Fig. 9.5b. CSG\(_y\)(H\(_2\)).
Here from Fig. 9.6 we have that

\[ H_2 = c T_1 \otimes T_2, \]

and so we may now even deduce that:

\[ H_2 \in \text{CSR} \]

(Actually \( H_2 \in \text{"2PL" \subset CSR} \) from Section 2.2.7).

![Diagram](image-url)  

Fig. 9.6. CSG(H).
9.1.2 SKELETON-DATABASES WITH WANDER-TRANSACTIONS

Now we shall look at some interesting points in the space between the two extreme points given in the previous section for the designated class ASR*.

GENERAL GUIDELINE

For the treatment of class ASR* we need to introduce a slightly new taxonomy for the separate items in a skeleton-database.

Thus let us look at the following different naming schemes for items in a distributed database:

\[ D = \]

\[
\begin{array}{cccccccc}
  x_{1,1} & x_{1,2} & x_{1,3} & \cdots & x_a & x_b & x_c & \cdots & x \\
  x_{2,1} & x_{2,2} & x_{2,3} & \cdots & y_a & y_b & y_c & \cdots & y \\
  x_{3,1} & x_{3,2} & x_{3,3} & \cdots & z_a & z_b & z_c & \cdots & z \\
  \vdots  & \vdots  & \vdots  & \ddots & \vdots  & \vdots  & \vdots  & \ddots & \vdots  \\
\end{array}
\]  

(Eq. 9.26)

Initially, the scheme to the left coincides with the old original view of a distributed database from Fig. 3.1 in Section 3.1.2.

The first index specifies a site within the global database, while the second index specifies an item within the local site.

Further, the scheme in the middle constitutes the new general view appropriate for a skeleton-database as outlined in Section 6.2.

Here \( x, y \) and \( z \) specify different occurrences of sites, while \( a, b \) and \( c \) specify different types of items.

So \( x_a, x_b \) and \( x_c \) represent different items (articles) existing (available) at a unique site (department store), and they correspond to an intra-site problem.

And \( x_a, y_a \) and \( z_a \) represent a unique item (article) existing (available) at different sites (department stores) - i.e. we talk about separate item-variants, and they correspond to an inter-site problem. Observe the difference between separate item-variants and respectively separate item-versions as mentioned in Section 2.3.2 and separate item-copies as discussed in Section 3.1.3. Item-variants are separate physical occurrences of a common logical item-type.

Finally, the scheme to the right constitutes a new special view appropriate for a skeleton-database containing only one unique item - though with several separate item-variants.
Thus $x$, $y$ and $z$ represent the unique item (article) existing (available) at different sites (department stores).

Such a simplified view is needed to explain the basic ideas of class ASR*. Hence this special view will be used in the coming sections, and we will only return to the general view in Sections 9.2.1, 9.5 and 9.6. Or paraphrased; we will basically emphasise the inter-site problem. (But we will allow a wander-transaction to acquire more than one of the separate item-variants corresponding to the unique item).

A combined illustration of the two new views is given in Fig. 9.7.

![Diagram](image)

Fig. 9.7. Correspondence between different non-interrelated Database Items.

It is obvious that the local databases of each separate site (department store) do not merely constitute the results from several horizontal fragmentation operations on a global database, see Section 3.1.3 (anticipating a relational data model). The items of corresponding fragments (possibly of several items each) in such a distributed database only have the same logical type. While the items of corresponding parts (necessarily of one item each) in our skeleton-database both have the same logical type and represent variants of the same physical entity.

For the development of correctness criteria for class ASR*, let us also outline a paradigm corresponding to our discussions in Sections 8.1.3 and 8.1.4.

- The inherent flexibility in the skeleton-database concept as presented in Section 6.2 - i.e. corresponding to the lack of some integrity constraints, represents some degrees-of-freedom with respect to an eventual correctness criterion for the use of this type of distributed database. This corresponds to possibility ii) in Section 6.3, and it was referred to as the special case in Fig. 8.1 from Section 8.1.3.

- The inherent logic in the wander-transaction concept as presented in Section 7.2 - i.e. corresponding to the existence of some overall semantics information, reduces the degrees-of-freedom with respect to the actual correctness criterion for the use of this type of skeleton-database. This corresponds to possibility ii) in Section 7.3, and the
reduced degrees-of-freedom will be mirrored by a resulting solution span between classes CMX and CSR from the previous section. In accordance with the classification in Section 8.3 we will name the correctness criteria corresponding to the CMX, CSR and ASR* classes of schedules respectively MUX, SER and ASER*, and this is illustrated in Fig. 9.8.

(The reason for using CSR and not FSR as a lower bound will be given in Section 9.2.1, and the motivation for using CMX as an upper bound is the need for still implementing atomic actions per single item).

Fig. 9.8: Relationship between different Classes of Schedules (and between the corresponding Correctness Criteria).
DEGREES-OF-FREEDOM

As pointed out in the previous section

\[ H \in \text{CMX} \iff \text{All WR-RW-WW}_x(H) - x \in D - \text{Partial Orders (Separately)}, \]

while

\[ H \in \text{CSR} \iff \text{WR-RW-WW}(H) \text{ Partial Order}, \]

or equivalent

\[ H \in \text{CSR} \iff \text{All WR-RW-WW}_x(H) - x \in D - \text{Consistent Partial Orders}. \]

So while an \( H \in \text{CMX} \) may have several mutually inconsistent serialization orders associated with each item separately, an \( H \in \text{CSR} \) must have one consistent serialization order associated with all items together.

Thus the possible points in the above-mentioned solution span for class ASR* must correspond to the space between:

\[ \forall x \in D \ [\text{WR-RW-WW}_x(\text{Any Schedule}) \text{ Partial Order}] \ [\text{MUX}] \]

&

\[ \text{WR-RW-WW(Any Schedule) Partial Order} \ [\text{SER}] \]

First, even though in the end we shall employ correctness criteria at least as strong as MUX - corresponding to class CMX, it is both illustrating and useful to try to dissect criterion MUX. This might be called a local - or per single item - freedom. We will here base our investigation on the set of binary relations given in Eqs. 9.1 to 9.7 in Section 9.1.1. In Table 9.1 we have shown the resulting options for a specific item \( x \) when we look at the different ways the three basic (local) binary relations \( WR_x, RW_x \) and \( WW_x \) may be naturally combined. (The statements about the different binary relations are of course understood to apply for any schedule).

The three classes at the top of the table are those corresponding to the weakest criteria - short of the universal class A corresponding to no criteria at all, while the class at the bottom is that corresponding to the strongest criterion (i.e. "MUX_x").

(As usual, the requirement that two or three simple binary relations are to be consistent partial orders is equivalent to require that one compound binary relation - effectively designated through the name of the corresponding class - is to be a partial order).

The hypothetical corresponding local criteria - i.e. per single item, require the same kind of restrictions put on each separate item - \( x \in D \), and hence we need to introduce:

\[ C_{L: \text{name}} = \bigcap_{x \in D} C_{\text{name}_x} \]  

(Eq. 9.27)
Thus for the sake of completeness, we get:

\[
\begin{align*}
C_L: WR &= \bigcap_{x \in D} C_{WR_x} \\
C_L: RW &= \bigcap_{x \in D} C_{RW_x} \\
C_L: WW &= \bigcap_{x \in D} C_{WW_x} \\
C_L: WR, RW &= \bigcap_{x \in D} C_{WR, RW_x} \\
C_L: WR, WW &= \bigcap_{x \in D} C_{WR, WW_x} \\
C_L: RW, WW &= \bigcap_{x \in D} C_{RW, WW_x} \\
C_L: WR-RW &= \bigcap_{x \in D} C_{WR-RW_x} \\
C_L: WR-WW &= \bigcap_{x \in D} C_{WR-WW_x} \\
C_L: RW-WW &= \bigcap_{x \in D} C_{RW-WW_x} \\
C_L: WR, RW, WW &= \bigcap_{x \in D} C_{WR, RW, WW_x} \\
C_L: WR-RW, WW &= \bigcap_{x \in D} C_{WR-RW, WW_x} \\
C_L: WR-WW, RW &= \bigcap_{x \in D} C_{WR-WW, RW_x} \\
C_L: RW-WW, WR &= \bigcap_{x \in D} C_{RW-WW, WR_x} \\
C_L: WR-RW-WW &= \bigcap_{x \in D} C_{WR-RW-WW_x}
\end{align*}
\]
From this again (and from Table 9.1) we see that:

- \( CM_X = C_{WR-RW-WW_X} \) \hspace{1cm} (Eq. 9.28)

\&

- \( CM_X = C_{L:WR-RW-WW} \) \hspace{1cm} (Eq. 9.29)

Not all the classes shown above are different. In the next paragraph we will carry out an initial investigation of the eventual effects of the differences in their specifications, and in Section 9.2.2 we shall include a lengthy analysis of the relationships between the classes.

Second, it is both necessary and interesting to dissect criterion SER the same way. This may be called a global - or per multiple item - freedom. We must now base our investigation on the set of binary relations given in Eqs. 9.14 to 9.20 in Section 9.1.1. In Table 9.2 we have shown the resulting options when we once more look at the different ways the three basic (global) binary relations WR, RW and WW may be naturally combined. (The statements about the different binary relations are of course again understood to apply for any schedule).

Table 9.2. Scheduling Options per Multiple Item.

<table>
<thead>
<tr>
<th>Class</th>
<th>Criterion</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C_{GWR} )</td>
<td>WR  Partial Order</td>
</tr>
<tr>
<td>( C_{GWR} )</td>
<td>RW  Partial Order</td>
</tr>
<tr>
<td>( C_{GWR} )</td>
<td>WW  Partial Order</td>
</tr>
<tr>
<td>( C_{GWR,WR} )</td>
<td>WR &amp; RW Partial Orders</td>
</tr>
<tr>
<td>( C_{GWR,WW} )</td>
<td>WR &amp; WW Partial Orders</td>
</tr>
<tr>
<td>( C_{GRR,WR} )</td>
<td>WR &amp; RW Consistent Partial Orders</td>
</tr>
<tr>
<td>( C_{GRR,WW} )</td>
<td>WR &amp; WW Consistent Partial Orders</td>
</tr>
<tr>
<td>( C_{GRR,WW,WR} )</td>
<td>WR &amp; RW &amp; WW Partial Orders</td>
</tr>
<tr>
<td>( C_{GRR,WW,WR} )</td>
<td>Consistent Partial Orders and WW Partial Order</td>
</tr>
<tr>
<td>( C_{GRR,WW,WR} )</td>
<td>Consistent Partial Orders and WR Partial Order</td>
</tr>
<tr>
<td>( C_{GRR,WW,WR} )</td>
<td>WR &amp; WW Consistent Partial Orders</td>
</tr>
</tbody>
</table>

Once more, the three classes at the top of the table are those corresponding to the weakest criteria - short of the universal class A corresponding to no criteria at all, while the class at the bottom is that corresponding to the strongest criterion (i.e. CSR).

(Again as usual, the requirement that two or three simple binary relations are to be consistent partial orders is equivalent to require that one compound binary relation - effectively designated through the name of the corresponding class - is to be a partial order).
For the actual local criterion - i.e. MUX, the corresponding total criteria - i.e. the combination of a global criterion per multiple item and the local criterion per single item, requires the introduction of:

\[- C_{\text{name}} = C_{G: \text{name}} \cap C_{L: \text{WR-RW-WW}}\]  

(Eq. 9.30)

\[\text{Thus for the sake of completeness, we get:}\]

\[C_{\text{WR}} = C_{G: \text{WR}} \cap C_{L: \text{WR-RW-WW}}\]
\[C_{\text{RW}} = C_{G: \text{RW}} \cap C_{L: \text{WR-RW-WW}}\]
\[C_{\text{WW}} = C_{G: \text{WW}} \cap C_{L: \text{WR-RW-WW}}\]
\[C_{\text{WR,RW}} = C_{G: \text{WR,RW}} \cap C_{L: \text{WR-RW-WW}}\]
\[C_{\text{WR,WW}} = C_{G: \text{WR,WW}} \cap C_{L: \text{WR-RW-WW}}\]
\[C_{\text{RW,WW}} = C_{G: \text{RW,WW}} \cap C_{L: \text{WR-RW-WW}}\]
\[C_{\text{WR-RW}} = C_{G: \text{WR-RW}} \cap C_{L: \text{WR-RW-WW}}\]
\[C_{\text{WR-WW}} = C_{G: \text{WR-WW}} \cap C_{L: \text{WR-RW-WW}}\]
\[C_{\text{RW-WW}} = C_{G: \text{RW-WW}} \cap C_{L: \text{WR-RW-WW}}\]
\[C_{\text{WR,RW,WW}} = C_{G: \text{WR,RW,WW}} \cap C_{L: \text{WR-RW-WW}}\]
\[C_{\text{WR-RW,WW}} = C_{G: \text{WR-RW,WW}} \cap C_{L: \text{WR-RW-WW}}\]
\[C_{\text{WR-WW,RW}} = C_{G: \text{WR-WW,RW}} \cap C_{L: \text{WR-RW-WW}}\]
\[C_{\text{RW-WW,WR}} = C_{G: \text{RW-WW,WR}} \cap C_{L: \text{WR-RW-WW}}\]
\[C_{\text{WR-RW-WW}} = C_{G: \text{WR-RW-WW}} \cap C_{L: \text{WR-RW-WW}}\]

From this again (of course) we see that:

\[\text{CSR} = C_{\text{WR-RW-WW}} = C_{G: \text{WR-RW-WW}}\]  

(Eq. 9.31)

Once more not all the classes shown above are different. In the next paragraph we will also carry out an initial investigation of the eventual effects of the differences in their specifications, and in
Section 9.2.1 plus Sections 9.4.2 and 9.4.3 we shall also include analyses of the relationships between the classes. Effectively some of these classes will be treated further as they are necessary for our system(s), and yet other of these classes will be commented upon as they are interesting extra variants.

Let us here just indicate the most direct consequences of eventually requiring that there is to be a partial order for each basic global binary relation WR, RW or WW.

- **WR(Any Schedule) Partial Order**

  implies in (the simplest case) that:

  \[
  W_2(x) \rightarrow R_1(x) \rightarrow 1 \quad W_1(y) \rightarrow R_2(y) \quad [x \in D, y \in D]
  \]

  (With respect to the meaning of such patterns, refer to the comments concerning Fig 2.13 in Section 2.2.4).

  Thus after a first transaction \( T_1 \) has read one part (i.e. \( x \)) of a second transaction \( T_2 \)'s result, then \( T_2 \) again must not read another part (\( y \)) of \( T_1 \)'s result.

  In case this could happen, either \( T_1 \) would have to be forced to pause (wait) its result-setting, or \( T_2 \) would have to stop (abort) its picture-building.

- **RW(Any Schedule) Partial Order**

  implies (in the simplest case) that:

  \[
  R_2(x) \rightarrow W_1(x) \rightarrow 1 \quad R_1(y) \rightarrow W_2(y) \quad [x \in D, y \in D]
  \]

  Thus after a first transaction \( T_1 \) has overwritten one part (i.e. \( x \)) of a second transaction \( T_2 \)'s picture, then \( T_2 \) again must not overwrite another part (\( y \)) of \( T_1 \)'s picture.

  In case this could happen, either \( T_1 \) would have to be forced to pause (wait) its picture-building, or \( T_2 \) would have to stop (abort) its result-setting.

- **WW(Any Schedule) Partial Order**

  implies (in the simplest case) that:

  \[
  W_2(x) \rightarrow W_1(x) \rightarrow 1 \quad W_1(y) \rightarrow W_2(y) \quad [x \in D, y \in D]
  \]

  Thus after a first transaction \( T_1 \) has overwritten one part (i.e. \( x \)) of a second transaction \( T_2 \)'s result, then \( T_2 \) again must not overwrite another part (\( y \)) of \( T_1 \)'s result.

  In case this could happen, either \( T_1 \) would have to be forced to pause (wait) its result-setting, or \( T_2 \) would have to stop (abort) its result-setting.
Similar statements may also be given for the basic local binary relation $WR_x$(Any Schedule), $RW_x$(Any Schedule) or $WW_x$(Any Schedule) - $x \in D$. But such a statement will only be appropriate for $RW_x$(Any Schedule) - as both $WR_x$(Any Schedule) and $WW_x$(Any Schedule) must be partial orders per definition. Remember from Section 2.2.4 both the restriction of maximum one read and/or one write per item per transaction and the prohibition against a read to succeed a write in order on a common item in a transaction.

However, when it comes to combined statements about two or three basic binary relations, both the global and local variants are interesting.
INITIAL INVESTIGATION

From now on it is both useful and necessary to distinguish between the two cases of read-before-write and not-read-before-write from Section 2.2.4.

The not-read-before-write case corresponds to the universal class A of all (legal) schedules:

\[-C_{\text{NR}} = A\]  \hspace{1cm} (Eq. 9.32a)

(In Section 4.2.1 this class was formally specified as there was a slight difference between all possible and all actual schedules according to the notation used in the two source-materials of Chapter 4).

The read-before-write case corresponds to the restricted class of all schedules where a write on a specific item by a certain transaction always must be preceded by a read on the same item by the same transaction:

\[-C_{\text{R}} = \{H \mid H \in A \land \]  \hspace{0.5cm} (Eq. 9.32b)

\[[[x \in D \land W_i(x) \in t_i] \Rightarrow [R_i(x) \in t_i \land R_i(x) \prec_t W_i(x)]]\}

(This specification is a direct application of Eq. 2.42 in Section 2.2.9).

The reason for employing such a case splitting will be given in Section 9.2.1.

Let us then review the two basic concurrency control problems encountered in Section 2.2.5 - i.e. the lost update problem and the inconsistent retrievals problem.

First, we will treat the read-before-write case.

In the two following schedules, a typical lost update situation occurs in $H_3$, while a typical inconsistent retrievals situation occurs in $H_4$.

$H_3 =$

$T_1: R_1(x),\ W_1(x)$
$T_2: R_2(x),\ W_2(x)$

$H_4 =$

$T_1: R_1(x),\ R_1(y)$
$T_2: R_2(x)W_2(x),\ R_2(y)W_2(y)$
(Effectively, $H_3$ equals the schedule in Fig. 2.14 from Section 2.2.5, and $H_4$ equals the schedule in Fig. 2.15 from the same section).

The CSG$_x$ of $H_3$ and the CSG of $H_4$ are given in respectively Fig. 9.9 and Fig. 9.10.

Second, we will treat the not-read-before-write case.

In the two following schedules, again a typical lost update situation occurs in $H_3^-$, while a typical inconsistent retrievals situation occurs in $H_4^-$.  

\[
H_3^- = \\
T_1: W_1(x) \\
T_2: R_2(x) \quad W_2(x)
\]

\[
H_4^- = \\
T_1: R_1(x) \quad R_1(y) \\
T_2: W_2(x)W_2(y)
\]

(Once more effectively, $H_3^-$ equals the schedule in Fig. 2.14 from Section 2.2.5, and $H_4^-$ equals the schedule in Fig. 2.15 from the same section).

The CSG$_x$ of $H_3^-$ and the CSG of $H_4^-$ are given in respectively Fig. 9.11 and Fig. 9.12.
From the above graphs we may deduce some simple results concerning the involved binary relations. These are summarized in Table 9.3.

Table 9.3. Characterization of certain Problem Cases.

<table>
<thead>
<tr>
<th>System Type</th>
<th>Read-before-write</th>
<th>Not-read-before-write</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Local/Global Problem Area</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I) (Basic)</td>
<td>RW [-WW]</td>
<td>RW-WW</td>
</tr>
<tr>
<td>Lost Update</td>
<td>Not Partial Order</td>
<td>Not Partial Order</td>
</tr>
<tr>
<td>II) Basic</td>
<td>WR-RW</td>
<td>WR-RW</td>
</tr>
<tr>
<td>Inconsistent</td>
<td>Not Partial Order</td>
<td>Not Partial Order</td>
</tr>
<tr>
<td>Retrieval</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Later analyses - carried out in Section 9.2.2 - will show that the basic lost update problem is the only problem possibly occurring locally - i.e. per single item - in both the read-before-write and not-read-before-write cases.
Hence the simple results in Table 9.3 concerning the local problem area will have general validity.

But, later discussions - summarized in Section 9.4.1 - will show that the basic inconsistent retrievals problem is not the only problem possibly occurring globally - i.e. per multiple item - in neither the read-before-write nor not-read-before-write cases.

Hence the simple results in Table 9.3 concerning the global problem area will be too coarse-grained.

Let us here just indicate the more obvious effects of our simple results so far.

First, we will look at the local and global problem areas separately.

1) Local Effects (i.e. per Single Item)

a) Read-before-write Case:

From

\[
\text{WR-RW-WW}_x(\text{Any Schedule}) \text{ Partial Order } [x \in D]
\]

excluding

RW

makes

Lost Update [on x]

possible.

b) Not-read-before-write Case:

From/in

\[
\text{WR-RW-WW}_x(\text{Any Schedule}) \text{ Partial Order } [x \in D]
\]

either totally excluding

RW or WW

or even allowing

RW and WW Non-Consistent

makes

Lost Update [on x]

possible.
2) Global Effects (i.e. per Multiple Item)

a)+b) Read-before-write and Not-read-before-write Cases:

From/in

WR-RW-WW(Any Schedule) Partial Order

either totally excluding

WR or RW

or even allowing

WR and RW Non-Consistent

makes

Inconsistent Retrievals

possible.

Second, we will comment on the combination of the local and global problem areas.

The combined effects of any local criterion and any global criterion are straightforward to deduce from the above stated facts.

However note that a global criterion (per multiple item) also induces a local criterion (per single item) per definition. Thus a local criterion could not be weaker than a global criterion, but the local criterion may be stronger than the global one.

As an example, [Gray76] mentions three analysis notions from which three interesting class definitions may be derived. According to the naming of Section 2.3.2, these will be SGT-type generalizations of the (2PL- or) LUE-type classes $C_1$, $C_2$ and $C_3$ of Section 5.3. With reference to Table 9.2 and Eq. 9.31 we have (when we also include the class $C_0$ of the same section):

- $C_0^\prime = C_0$  \hspace{1cm} (Eq. 9.33a)

- $C_1^\prime = C_{G:WW}$  \hspace{1cm} (Eq. 9.33b)

- $C_2^\prime = C_{G:WR-WW}$  \hspace{1cm} (Eq. 9.33c)

- $C_3^\prime = C_{G:WR-RW-WW} = CSR$  \hspace{1cm} (Eq. 9.33d)
Note that there is no division between a global criterion and a local
criterion. Hence the local criterion is not stronger than the global
one. This stems from the corresponding notions being defined for a
centralized setting rather than a distributed setting. Further observe
that the stated class-definitions will all be different from the
(global part of the) class-definitions that we will end up with. (See
Section 9.2.1 for the investigation leading up to the specifications
of our correctness criteria). The relations between the classes given
here and our classes will be analyzed in depth in Section 9.4.1. Note
also that all transactions of a schedule being a member in class \( C_i \)
(except \( C_0' \)) execute at the same "level" \( i \); i.e. according to the same
concurrency control and recovery criteria. This is in contrast to
class \( C_i \) of Section 5.3 where each transaction executes at at least
level \( i \).

It is easy to deduce the effects with respect to possible concurrency
control problems for the classes \( C_0' - C_3' \) from the above statements
and the above equations. They will be the same as for the classes \( C_0 - C_3 \) with the ReLock alternative, as stated in Table 5.6 in Section 5.3.
This applies to both the not-read-before-write and read-before-write
cases. Thus the class-division given here is still fairly coarse-
grained on concurrency control. The underlying notions still seem to be
less concurrency control oriented. Effectively even going from
SerializationTesting via 2PhaseLocking to LockUntilEnd (see
Section 2.3.2) induces no gain with respect to any of the treated
concurrency control problems.

As another example, let us refer to the multiversion serializability
mentioned in Section 8.1.5. The essential point is that a schedule
which is non-serializable with only one version per item, may still be
serializable with several versions per item. With reference to the
classes CSR and VSR\(_0\) in Section 2.2.6, let us name the class of schedules
which are conflict serializable with several versions per item or old view serializable with several versions per item
respectively MCSR and MVSR\(_0\). Again with reference to Table 9.2 and Eq.
9.31 we have (when we assume only one version per item):

\[-\text{MCSR} = C_{G:RW} \quad \text{(Eq. 9.33e)}\]

Once more there is no division between a global criterion and a local
criterion, and the stated class-definition will be different from the
(global part of the) class-definitions that we will end up with. Further we have:

- \( \text{MVSR}_0 = \text{MCSR} \) (for the Read-before-write Case)
- \( \text{MVSR}_0 \supset \text{MCSR} \) (for the Not-read-before-write Case)

The relations between the classes given here and our classes will also
be analyzed in Section 9.4.1.

To round off, an inclusion of the binary relations \( \text{WR}(H) \) and \( \text{RW}(H) \) in
a (global) criterion might be said to relate to view acquisition in a
schedule \( H \), while an additional inclusion of the binary relation \( \text{WW}(H) \)
in the (global) criterion might be said to relate to even consistency
preservation for a schedule \( H \) (see Section 2.2.6).
9.2 Criteria for Alternative-to-Serializability

In this section we will fix two points in the investigated solution span for the class ASR*. The first point applies to the read-before-write case, and the corresponding class will be named \( A_{R}SR \). The second point applies to the not-read-before-write case, and the corresponding class will be named \( A_{NR}SR \).

9.2.1 General Discussion and Specifications

Here we shall carry out the general investigation leading up to the specifications of correctness criteria for the two above-mentioned classes \( A_{R}SR \) and \( A_{NR}SR \).

Main Approach

With reference to Table 9.2 in the previous section, the classes that we are going to employ for our systems will correspond to that from the common total criterion

\[ WR-RW-WW(\text{Any Schedule}) \text{ Partial Order} \]

one or two binary relation part(s) will be excluded. Thus we will not exploit the option to allow one or two partially ordered binary relation part(s) to be non-consistent with the other partially ordered binary relation part(s). But in Sections 9.4.2 and 9.4.3 we will discuss such corresponding extra variants as actual intermediaries between our choices and the normal choice indicated above.

Besides, our discussion and specifications will lead to two separate ways of allowing

\[ WR-RW-WW(\text{Some Schedule}) \text{ Non-Partial Order}, \]

and still having

\[ \text{Our-Relation(Any Schedule) Partial Order}. \]

First, cycles occurring in the corresponding CSG-graph may be avoided in an alternative Our-graph by seeing that a subcycle vanishes. This is exemplified in Fig. 9.13.

Of course deleting some

\[ T_{j} \rightarrow T_{k} \text{ Arc}, \]

requires that there is no additional

\[ T_{j} \rightarrow \ldots \rightarrow T_{k} \text{ Path}. \]

(The arc-types indicated by arc-labels will be discussed later).
Fig. 9.13. A Cycle from which One (or more) Arc(s) will be removed.

Second, cycles occurring in the corresponding CSG-graph may be avoided in an alternative Our-graph by seeing to that a total cycle vanishes. This is exemplified in Fig. 9.14.

Of course deleting some

\[ T_j \rightarrow T_i \text{ & } T_i \rightarrow T_j \text{ Cycle,} \]

requires that there is no additional

\[ T_j \rightarrow \ldots \rightarrow T_i \text{ & } T_i \rightarrow \ldots \rightarrow T_j \text{ Cycle.} \]

(The arc-types indicated by arc-labels will again be discussed later).

Both the indicated ways will be applied for both the read-before-write and not-read-before-write cases.
PRE-ANALYSIS

Two larger paragraphs whose results are required before we may turn to the discussion and specifications of classes \( A_{SR} \) and \( A_{WR,SR} \), have been included separately in Section 9.2.2.

Let us extract the main consequences of their results.

First, refer to the paragraph called **local/global-relations coherence**.

In the read-before-write case we may - with reference to Table 9.2 in Section 9.1.2 - deduce that it adds no freedom to delete from any global criterion at least as strong as corresponding to the class

\[ C_G: \ldots RW-WW \ldots \]

the requirement that the partially ordered binary relation \( WW(\text{Any Schedule}) \) has to be consistent with some other partially ordered binary relation(s), ending up with:

\[ C_G: \ldots RW,WW \ldots \]

Further it still adds no freedom even to delete the requirement that the binary relation \( WW(\text{Any Schedule}) \) has to be partially ordered, ending up with:

\[ C_G: \ldots RW \ldots \]

(And the same will apply to a binary relation \( WW_x(\text{Any Schedule}) \) with respect to a local criterion - still in the read-before-write case only, but this time with reference to Table 9.1 in Section 9.1.2).

The reason is that any write-write relationship in a binary relation \( [WR-]RW-WW(\text{The Schedule}) \) or \( [WR-]RW-WW_x(\text{The Schedule}) \) - \( x \in D \) - will be mirrored by a corresponding read-write relationship in the read-before-write case (see Eq. 9.67).

Thus in the read-before-write case we have the following sets of equalities between the globally oriented classes of Table 9.2:

\[ C_G:RW-WW = C_G:RW,WW = C_G:RW \quad \text{(Eq. 9.34)} \]

\[ C_G:RW-WW,WR = C_G:WR,RW,WW = C_G:WR,RW \quad \text{(Eq. 9.35)} \]

\[ C_G:WR-RW-WW = C_G:WR-RW,WW = C_G:WR-RW \quad \text{(Eq. 9.36)} \]

(And we will have the same sets of equalities between the corresponding locally oriented classes - still in the read-before-write case, but this time with reference to Table 9.1).
Second, refer to the paragraph called *local-criteria coherence*.

In the *read-before-write case* we may - with reference to the locally oriented variants of Eq. 9.36 - even deduce the following class equalities for each \( x \in D \):

\[
- \quad C_{WR-RW-\text{WW}}^x = C_{RW}^x \\
& \quad \text{(Eq. 9.37)}
\]

\[
- \quad C_{L:WR-RW-\text{WW}} = C_{L:RW} \\
& \quad \text{(Eq. 9.38)}
\]

So comparing Eq. 9.37 with Eq. 9.36, we see that we may go further with respect to class equalities locally than globally.

Even in the *not-read-before-write case* we may - for locally oriented variants - deduce the following class equalities for each \( x \in D \):

\[
- \quad C_{WR-RW-\text{WW}}^x = C_{RW-\text{WW}}^x \\
& \quad \text{(Eq. 9.39)}
\]

\[
- \quad C_{L:WR-RW-\text{WW}} = C_{L:RW-\text{WW}} \\
& \quad \text{(Eq. 9.40)}
\]

The difference between Eq. 9.39 and Eq. 9.37 of course stems from the above-mentioned reason leading to the existence of Eqs. 9.34 to 9.36.

Thus in both the read-before-write and not-read-before-write cases, to make any schedule \( H \) at least \( x \)-conflict serializable, it is not required even to assure

\[
\text{WR-RW-\text{WW}}_x(H) \quad \text{Partial Order} \quad [x \in D].
\]

Observe that the facts stated here fully support the per single item part of the initial investigation in Section 9.1.2.
READ-BEFORE-WRITE CASE

This case is to have our primary attention, as it corresponds directly and naturally to the application of wander-transactions in skeleton-databases. Thus the results of this section will concern our main alternative system. (See the discussion of case grouping carried out in the next section and summarized later in this section).

First, we will investigate the local requirements; i.e. per single item.

After a transaction has retrieved a single item, this item should not be changed through an update of another transaction until the given transaction either actually updates the specific item itself or effectively relinquishes its right to do so. If we did not assure this, we would allow a transaction’s basis with respect to a local decision to be invalidated by another transaction.

Such a restriction must of course apply to all transactions and all items. Effectively this means that we will not allow any lost update problem to occur. Basically we are thus requiring (conflict) mutual exclusive executions of transactions (see Section 9.1.1).

So with reference to Eqs. 9.27 and 9.29 in Section 9.1.2 our local choice is:

\[ C_L:WR-RW-WW \quad (= \text{CMX}) \]

Second, we will investigate the global requirements; i.e. per multiple item.

When a transaction retrieves several items, the values of these items should not have to correspond to a single database state corresponding to a (conflict) serializable execution of transactions (see Section 9.1.1). The reason is the lack of integrity constraints in a skeleton-database (see Section 6.3).

Basically we may thus allow some "inconsistent" retrievals problems to occur. Effectively, though not allowing a transaction’s basis with respect to a local decision to be invalidated by one or more other transactions, we will allow a transaction’s bases with respect to a global decision to be influenced by one or more other transactions.

We must now turn to the existence of overall semantics information for a wander-transaction (see Section 7.3).
As mentioned earlier in this section, the classes that we are going to employ for our systems will correspond to that from the common global criterion

WR-RW-WW(Any Schedule) Partial Order

one or two binary relation part(s) will be excluded.

From one of our earlier findings - represented as Eq. 9.36 - we know that for the read-before-write case this global criterion is equal to requiring:

WR-RW(Any Schedule) Partial Order

Thus excluding only the WW-part does not gain us any freedom at all. Should we then also exclude the WR-part or the RW-part?

Let us start by looking at an eventual exclusion of the WR-part.

This would leave us with the following resulting global criterion:

RW(Any Schedule) Partial Order

The most direct consequences of such a requirement was illustrated in Section 9.1.2:

- RW(Any Schedule) Partial Order

implies (in the simplest case) that:

\[ R_2(x) \rightarrow W_1(x) \Rightarrow 1 R_1(y) \rightarrow W_2(y) \quad [x \in D, \ y \in D] \]

Thus after a first transaction \( T_1 \) has overwritten one part (i.e. \( x \)) of a second transaction \( T_2 \)'s picture, then \( T_2 \) again must not overwrite another part (\( y \)) of \( T_1 \)'s picture.

Enforcing such a restriction effectively means that for two transactions (in the simplest case), we would definitely not allow each one to make and implement a decision with respect to an item (i.e. acquiring the corresponding article through a write) which the other eventually could make a decision with respect to (i.e. having checked the corresponding article through a read), but very well might not want to make and implement a decision with respect to (i.e. not acquiring the corresponding article through a write). Remember that a transaction normally checks several articles, but acquires only one.

Further, by not enforcing such a restriction, we would still not create any difficulties in the eventual situation where both transactions actually do want to make and implement a common decision with respect to at least one of the two items (i.e. both acquiring at least one of the two corresponding articles through a write). The local criterion will catch such a situation.
Let us continue by looking at an eventual exclusion of the RW-part. This would leave us with the following resulting global criterion:

\textbf{WR(Any Schedule) Partial Order}

The most direct consequences of such a requirement was again illustrated in Section 9.1.2:

- \textbf{WR(Any Schedule) Partial Order}

implies in (the simplest case) that:

\[ W_2(x) \rightarrow R_1(x) \rightarrow 1 \quad W_1(y) \rightarrow R_2(y) \quad [x \in D, y \in D] \]

Thus after a first transaction \( T_1 \) has read one part (i.e. \( x \)) of a second transaction \( T_2 \)'s result, then \( T_2 \) again must not read another part (i.e. \( y \)) of \( T_1 \)'s result.

Enforcing such a restriction effectively means that for two transactions (in the simplest case), we would definitely not allow each one having made and implemented a decision with respect to an item (i.e. having acquired the corresponding article through a write) upon whose result the other actually would partially base its further decision with respect to this item-type (i.e. checking the corresponding article through a read).

Hence, by not enforcing such a restriction, we could eventually create some odd situations. Two or more transactions might effectively be involved in a double or recursive "inconsistent" retrievals problem.

We definitely mean that the effects of an exclusion of the WR-part (i.e. leaving the RW-part included) are not sensible, while the effects of an exclusion of the RW-part (i.e. leaving the WR-part included) are quite sensible - for our system of wander-transactions in a skeleton-database.

Further, note the high dominance of reads over writes and the high dominance of late writes over early writes inherent in the semantics of wander-transactions. This makes a global requirement which corresponds to the binary relation RW being a partial order, even more limiting with regard to allowing possible race-situations for favourable offers. Likewise this makes a global requirement which corresponds to the binary relation WR being a partial order, even also suitable as a priority mechanism in controlling actual race-situations for favourable offers.

A global WR-order may thus be used in solving breaks with the local requirement per item-variant and/or per site. (See Section 9.6.2 for the generalized local requirement per site). When a local break is about to occur, the WR-order is consulted. If the involved wander-transactions (two or more) are ordered, any latest ordered wander-transaction is selected for abortion. While if they are not ordered, any one may be chosen for abortion. This implies that quickly making and implementing decisions with respect to items, gives high priority with respect to avoiding that you get selected for abortion.
Such a use of the binary relation WR as a priority mechanism also requires that it has to be maintained as a partial order. When a global break is about to occur, the wander-transaction whose write is part of the write-read conflict about to close a cycle in the graph corresponding to the WR-relation, is selected for abortion.

Maintaining a global WR-order may thus even be used to influence the local WR-RW-WW-order per item-variant and/or per site. This means that quickly making and implementing decisions with respect to items, even gives high priority with respect to getting hold of other attractive offers.

The abortion of a wander-transaction may be very fast. Recall from Section 7.2 that acquiring an offer implies decrementing a quantity and adding an identity to a purchase-list. Hence aborting a transaction basically implies incrementing quantities and deleting identities from purchase-lists. Further the rescheduling of a wander-transaction which has been unnecessarily aborted, will lead to the same results for the transaction. (Observe that there will be some cases of this type).

We will return to these points in Section 9.6.1.

From this the results for our global requirements will be:

\[ \text{WR}(\text{Any Schedule}) \text{ Partial Order} \]

So with reference to Table 9.2 in Section 9.1.2 our global choice is:

\[ C_G:WR \]

Then with reference to Eq. 9.30 in Section 9.1.2 our combined global and local choices are:

\[ C_{WR} = C_G:WR \cap C_L:WR-RW-WW \quad (\text{Eq. 9.41}) \]

Let us review the immediate consequences of our choices when we couple them with some more of our earlier findings.

Combining Eqs. 9.28 and 9.29 in Section 9.1.2 plus Eq. 9.41 in this section with Eqs. 9.37 and 9.38 still in this section, we get for the read-before-write case:

\[ \text{CMX}_x = C_{WR-RW-WW}_x = C_{RW}_x \quad (\text{Eq. 9.42}) \]

\[ \text{CMX} = C_L:WR-RW-WW = C_L:RW \quad (\text{Eq. 9.43}) \]

\[ \text{AR}_S = C_{WR} = C_G:WR \cap C_L:RW \quad (\text{Eq. 9.44}) \]
NOT-READ-BEFORE-WRITE CASE

This case is to have some secondary attention, as it opens the way for a natural and direct generalization/extension of the results for wander-transactions in a skeleton-database. Thus the results of this section will concern an extra alternative system. (Again see the discussion of case grouping carried out in the next section and summarized later in this section).

First, we will investigate the local requirements; i.e. per single item.

The arguments and conclusions are now exactly the same as those for the local requirements in the read-before-write case above.

So again with reference to Eqs. 9.27 and 9.29 in Section 9.1.2 our local choice is:

\[ C_{L:WR-RW-WW} \ (\equiv \ CMX) \]

Second, we will investigate the global requirements; i.e. per multiple item.

The arguments and conclusions still resemble to a certain degree those for the global requirements in the read-before-write case above.

However, note the following difference.

In the read-before-write case a global criterion

\[ WR-RW(\text{Any Schedule}) \text{ Partial Order} \]

equals the global criterion

\[ WR-RW-WW(\text{Any Schedule}) \text{ Partial Order}. \]

But in the not-read-before-write case the above equality does not apply.

Further, observe the following difference.

In the read-before-write case the decisions may be more isolated as the local criterion, i.e. (conflict) mutual exclusion, is fairly strong due to the existence of the read-before-write coupling. Thus it may be sufficient to only leave the WR-part as the global criterion.

But in the not-read-before-write case the decisions must be less isolated as the local criterion, i.e. (conflict) mutual exclusion, is fairly weak due to the lack of a read-before-write coupling. Thus it may be insufficient to only leave the WR-part as a global criterion.
Even so, observe that the lack of integrity constraints in a database like the skeleton-database type actually leaves the determination of a global criterion for transactions like the wander-transaction type to choice. Some choices may be better than others by analyses and discussions, but no choice will be more correct than another per definition.

We therefore want to look at the effects of an exclusion of only the WW-part (i.e. leaving both the WR-part and the RW-part included) as an extra add on to our system of wander-transactions in a skeleton-database.

From this the results for our global requirements will now be:

\[ \text{WR-RW(Any Schedule) Partial Order} \]

So again with reference to Table 9.2 in Section 9.1.2 our global choice is:

\[ C_{G:WR-RW} \]

Then with reference to Eq. 9.30 in Section 9.1.2 our combined global and local choices are:

\[ C_{\text{WR-RW}} = C_{G:WR-RW} \cap C_{L:WR-RW-WW} \quad (\text{Eq. 9.45}) \]

Let us again review the immediate consequences of our choices when we couple them with some of our earlier findings.

Combining Eqs. 9.28 and 9.29 in Section 9.1.2 plus Eq. 9.45 in this section with Eqs. 9.39 and 9.40 still in this section, we get for the not-read-before-write case:

\[ C_{\text{WR-RW}} = C_{G:WR-RW-WW} = C_{\text{RW-WW}} \quad (\text{Eq. 9.46}) \]

\[ C_{\text{WR-RW}} = C_{L:WR-RW-WW} = C_{L:RW-WW} \quad (\text{Eq. 9.47}) \]

\[ A_{\text{NR-SR}} = C_{\text{WR-RW}} = C_{G:WR-RW} \cap C_{L:RW-WW} \quad (\text{Eq. 9.48}) \]
POST-DISCUSSION

Note that the discussion and specifications of classes $A_{RSR}$ and $A_{NRSR}$ both constitute a combined exploitation of the possibility ii) from Section 6.3 and the possibility ii) from Section 7.3. Class $A_{RSR}$ even contains an additional exploitation of the possibility i) from Section 7.3.

Observe also that the above stated choices have relations to the multiple item part of the initial investigation in Section 9.1.2. This will be covered in depth in Sections 9.3.1 and 9.3.2.

Three interesting paragraphs which further evaluate the results from the discussion and specifications of classes $A_{RSR}$ and $A_{NRSR}$, have also been included separately in Section 9.2.2.

Let us repeat their main conclusions.

First, refer to the paragraph called model effects.

Only for the combination of the special single-action-multi-item model in Chapter 4 (and not for the general multi-action-single-item model in Chapter 2) with the read-before-write case is our correctness criterion less relevant. This means that the global part of the membership requirements for class $A_{RSR}$ is assured for any schedule without any synchronization in the single-action-multi-item model.

Then, refer to the paragraph called criteria basis.

CSR is formally not the only possible starting point for our new correctness criteria - corresponding to $A_{RSR}$ and $A_{NRSR}$ - in absolutely all cases. But to have combinable and comparable criteria, CSR is the natural choice.

(Practically we also have to take the efficiency aspects into consideration. As mentioned in Section 4.3 this may rule out anything but CSR as criteria basis even for the remaining cases where there still are theoretically other possibilities).

Last, refer to the paragraph called case grouping.

Our main emphasis has been and will still be on the special read-before-write case and its corresponding correctness criterion. But we have also put and will also continue to put some emphasis on an accompanied treatment of the general not-read-before-write case and its possible correctness criterion. And we do this even without yet being able to state the exact application area of the results or the eventual combination of the results with those of the main case in its corresponding application area; i.e. wander-transactions in a skeleton-database.
LOCALLY-RESULTING NOTIONS

We will now mention some relevant effects of our discussion and specifications.

Naming the class $CMX_x$ in the read-before-write and not-read-before-write cases respectively $CMX_x$ and $CMX_x$, Eqs. 9.42 and 9.46 give us:

- $CMX_x = CMX_x \cap C_R = C_{RW_x} \cap C_R \quad (Eq. 9.49)$
- $CMX_x = CMX_x \cap C_{NR} = C_{RW-WW_x} \cap C_{NR} \quad (Eq. 9.50)$

Thus in the read-before-write case, to test class membership in $CMX_x - x \in D$ - for a given schedule $H$, it suffices to make a directed graph $C_{RSG_x(H)}$ as:

- $V ( C_{RSG_x(H)} ) = \{ T_i | t_i \in h \} \quad (Eq. 9.51)$
- $A ( C_{RSG_x(H)} ) = \{ T_j \rightarrow T_i | R_j(x) \sim_h W_i(x) \} \quad (a)$

Then $H \in C_{RMX_x}$ if and only if $C_{RSG_x(H)}$ is acyclic.

Case a) is illustrated in Fig. 9.15a.

![Fig. 9.15a. Resulting contributions to $C_{RSG_x(H)}$ from different types of conflicts in $H$.](image)

Compare Eqs. 9.51 to 9.52 and Fig. 9.15a with Eqs. 9.12 to 9.13 and Fig. 9.1 in Section 9.1.1.

And in the not-read-before-write case, to test class membership in $CMX_x - x \in D$ - for a given schedule $H$, it suffices to make a directed graph $C_{NRSG_x(H)}$ as:

- $V ( C_{NRSG_x(H)} ) = \{ T_i | t_i \in h \} \quad (Eq. 9.53)$
- $A ( C_{NRSG_x(H)} ) = \{ T_j \rightarrow T_i | R_j(x) \sim_h W_i(x) \} \quad (a)$
- $U \{ T_j \rightarrow T_i | W_j(x) \sim_h W_i(x) \} \quad (b)$
Then $H \in C_{NR} MX_x$ if and only if $C_{NR} SG_x(H)$ is acyclic.

Cases a) and b) are both illustrated in Fig. 9.15b.

![Diagram](image)

Fig. 9.15b. Resulting contributions to $C_{NR} SG_x(H)$ from different types of conflicts in $H$.

Again compare Eqs. 9.53 to 9.54 and Fig. 9.15b with Eqs. 9.12 to 9.13 and Fig. 9.1 in Section 9.1.1 - and with Eqs. 9.51 to 9.52 and Fig. 9.15a in this section.

However we will still employ the name $CMX_x$ both for $C_R MX_x$ and $C_{NR} MX_x$ and still use $CSG_x$ graphs both for $C_R SG_x$'s and $C_{NR} SG_x$'s as we wish to avoid too many new concepts simultaneously.
GLOBALLY-RESULTING CONCEPTS

We will here evaluate the important effects of our discussion and specifications.

First, let us look at the read-before-write case.

We may specify our primary new equivalence notion reflecting effects on multiple items based on Eq. 9.44.

**Read-alternative equivalent schedules** is a Binary Relation $=_{ra}:

$$H =_{ra} H' \iff$$

- $h = h'$

and

- $\forall (T_j, T_i) [(T_j, T_i) \in WR(H) \iff (T_j, T_i) \in WR(H')]$ (Eq. 9.56)

Likewise we may specify our primary new serializability notion reflecting effects on multiple items.

**Read-alternative serializable schedules** is a Set $A_{rs}SR:

$$H \in A_{rs}SR \iff$$

- $H \in CMX$ (Eq. 9.57)

and

- $\exists H_s \in S [H =_{ra} H_s]$ (Eq. 9.58)

Thus $H \in A_{rs}SR$ if and only if $WR(H)$ and each $[WR-]RW[-WW]_x(H) - x \in D -$ all are partial orders separately.

To test class membership in $A_{rs}SR$ for a given schedule $H$, make a directed graph $A_{rs}SG(H)$ as:

- $V (A_{rs}SG(H)) = \{ T_i \mid t_i \in h \}$ (Eq. 9.59)

- $A (A_{rs}SG(H)) = \{ T_j \rightarrow T_i \mid \exists x \in D \ [W_j(x) <_h R_i(x)] \}$ (a) (Eq. 9.60)

Then $H \in A_{rs}SR$ if and only if $A_{rs}SG(H)$ and each $C_{(R)SG_x}(H) - x \in D$ - all are acyclic separately.
Case a) is illustrated in Fig. 9.16a. Observe that the existing read-before-write coupling here makes it possible to only include a write-read arc (i.e. of type a)) for the pair of any specific transaction reading a given item plus the last transaction having written this item before the reading takes place. This is enabled even though there are no write-write arcs.

![Diagram](image)

**Fig. 9.16a. Resulting contributions to $A_{SG}(H)$ from different types of conflicts in $H$.**

Note further that any topological sort of an acyclic $A_{SG}(H)$ corresponds to a serial schedule $H_s$ read-alternative equivalent to $H$. Thus the given system schedule $H$ may have several read-alternative serializations. (However remember that for any such serialization to be legal, it also has to be $x$-conflict serializable for each $x \in D$).

A comparison of Eqs. 9.55 to 9.58 with Eqs. 9.8 to 9.11 in Section 9.1.1 and with Eqs. 9.21 to 9.23 in the same section shows that for the read-before-write case:

$$CMX \supset A_{SR} \supset CSR$$

(This may also be seen from Eq. 9.31 in Section 9.1.2 and Eqs. 9.43 and 9.44).

Thus compare even Eqs. 9.59 to 9.60 and Fig. 9.16a (concerning $A_{SG}$) with Eqs. 9.24 to 9.25 and Fig. 9.2 (concerning $CSG$) in Section 9.1.1.

Second, let us look at the not-read-before-write case.

We may specify our secondary new equivalence notion reflecting effects on multiple items based on Eq. 9.48.

**Not-read-alternative equivalent schedules** is a Binary Relation $e_{nra}$:

$$H \sim_{nra} H' \text{ iff }$$

- $h = h'$ \hspace{1cm} (Eq. 9.61)

and

- $\forall (T_j, T_i) [(T_j, T_i) \in WR-RW(H) \iff (T_j, T_i) \in WR-RW(H')]$ \hspace{1cm} (Eq. 9.62)
Likewise we may specify our secondary new serializability notion reflecting effects on multiple items.

**Not-read-alternative serializable schedules** is a Set \( A_{NRSR} \):

\[
H \in A_{NRSR} \text{ iff } \quad \begin{align*}
- & \ H \in \text{CMX} \\
\text{and} & \\
- & \exists H_s \in S \ [H \nra H_s] \quad (\text{Eq. 9.63})
\end{align*}
\]

Thus \( H \in A_{NRSR} \) if and only if \( WR-RW(H) \) and each \( [WR-]RW-WW_x(H) - x \in D \) - all are partial orders separately.

To test class membership in \( A_{NRSR} \) for a given schedule \( H \), make a directed graph \( A_{NRSG}(H) \) as:

\[
- V \ (A_{NRSG}(H)) = \{T_i | t_i \in h\} \quad (\text{Eq. 9.65})
\]

\[
A \ (A_{NRSG}(H)) = \quad (\text{Eq. 9.66})
\]

\[
\begin{align*}
\{T_j \rightarrow T_i | \exists x \in D \ [W_j(x) <_h R_i(x)] \} & \quad (a) \\
\cup \ {T_j \rightarrow T_i | \exists y \in D \ [R_j(y) <_h W_i(y)]} & \quad (b)
\end{align*}
\]

Then \( H \in A_{NRSR} \) if and only if \( A_{NRSG}(H) \) and each \( C_{(NR)}SG_x(H) - x \in D \) - all are acyclic separately.

Cases a) and b) are both illustrated in Fig. 9.16b. Observe that the lacking read-before-write coupling here makes it necessary to actually include write-read arcs (i.e. of type a)) for the pairs of any specific transaction reading a given item plus all the transactions having written this item before the reading takes place - and include read-write arcs (i.e. of type b)) for the pairs of any specific transaction having read a given item plus all the transactions writing this item after the reading takes place. This is required because there are no write-write arcs.

![Fig. 9.16b. Resulting contributions to A_{NRSG}(H) from different types of conflicts in H.](image)

Note further that any topological sort of an acyclic \( A_{NRSG}(H) \) corresponds to a serial schedule \( H_s \) not-read-alternative equivalent to
H. Thus the given system schedule $H$ may have several not-read-alternative serializations. (However remember that for any such serialization to be legal, it also has to be $x$-conflict serializable for each $x \in D$).

A comparison of Eqs. 9.61 to 9.64 with Eqs. 9.8 to 9.11 in Section 9.1.1 and with Eqs. 9.21 to 9.23 in the same section again shows that for the not-read-before-write case:

$$CMX \supset A_{\text{NSR}} \supset CSR$$

(This may also be seen from Eq. 9.31 in Section 9.1.2 and Eqs. 9.47 and 9.48).

Thus compare even Eqs. 9.65 to 9.66 and Fig. 9.16b (concerning $A_{\text{NSG}}$) with Eqs. 9.24 to 9.25 and Fig. 9.2 (concerning CSG) in Section 9.1.1 - and with Eqs. 9.59 to 9.60 and Fig. 9.16a (concerning $A_{\text{SG}}$) in this section.
MAIN VISION

Let us review the ideas behind our discussion and specifications.

Not all transactions necessarily operate on all the items or even on the same items. And when some transactions do operate on the same item, not all of them necessarily act all the way by both reading and writing or even act the same way by either reading or writing.

So for any schedule \( H \), going from requiring

\[ \text{WR-RW-WW}(H) \text{ Partial Order} \quad [\text{Total}] \]

to requiring

\[ \text{WR}(H) / \text{WR-RW}(H) \text{ Partial Order} \quad [\text{Global}] \]

&

\[ \forall x \in D \left[ \text{WR-} \text{RW-} \text{-WW}_x(H) / \text{WR-} \text{RW- WW}_x(H) \text{ Partial Order} \right] \quad [\text{Local}] \]

may correspond to going from one total binary relation containing many transaction orderings to one global binary relation containing less transaction orderings plus several local binary relations each containing few transaction orderings.

The possibility for a binary relation to be a partial order naturally increases with a decreasing number of contributions.

Thus the chance of fulfilling the global plus local requirements usually is more probable than the chance of fulfilling the total requirement.

Actually, in those cases where the shrinking of the total binary relation into the global binary relation plus the projections of the total binary relation onto the local binary relations do not delete any transaction ordering(s) in full, we do not gain any extra freedom that might help us from a non-partial order totally to partial order(s) globally plus locally.

However, in those cases where the shrinking of the total binary relation into the global binary relation plus the projections of the total binary relation onto the local binary relations do delete some transaction ordering(s) in full, we do gain some extra freedom that may help us from a non-partial order totally to partial order(s) globally plus locally.
MAIN RESULTS

Our approaches may so far be summarized in Tables 9.4 and 9.5.

Table 9.4 indicates what kind of basic concurrency control problems will and will not be allowed to occur in the read-before-write and not-read-before-write cases.

<table>
<thead>
<tr>
<th>Problem Case</th>
<th>Read-before-write</th>
<th>Not-read-before-write</th>
</tr>
</thead>
<tbody>
<tr>
<td>I) (Basic) Lost Update</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>II) Basic Inconsistent Retrieval</td>
<td>Y</td>
<td>N</td>
</tr>
</tbody>
</table>

N means: No, the problem case is not allowed to occur
Y means: Yes, the problem case is allowed to occur

As mentioned in the previous section, in Sections 9.3.1 and 9.3.2 we will make our results more fine-grained through analyses and discussions of several per multiple item problems - out of which the basic inconsistent retrievals problem is only one example. (The basic lost update problem is still the only per single item problem - to be shown in the next section). Then it will be shown that non-serializable situations may happen for both the read-before-write and not-read-before-write cases.

Table 9.5 illustrates what kinds of synchronization will be applied in the read-before-write and not-read-before-write cases.

Thus compare Table 9.5 with Table 9.3 in Section 9.1.2.

Requiring, as we do in the read-before-write case,

\[ \text{WR(H) Partial Order} \; \text{[Global]} \]
\&
\[ \forall x \in D \; \text{[[WR-JRW][-WW]]}_x(H) \; \text{Partial Order} \; \text{[Local]} \]

means that for each item where there is some WR-relationship(s), the corresponding local serialization order part must be consistent with the global serialization order. But for each item where there is no WR-relationship(s), the corresponding local serialization order may be non-consistent with the global serialization order.
Table 9.5. Characterization of chosen Scheduling Criteria.

<table>
<thead>
<tr>
<th>System Type</th>
<th>Read-before-write</th>
<th>Not-read-before-write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global/Local</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheduling Area</td>
<td></td>
<td></td>
</tr>
<tr>
<td>II) Per</td>
<td>WR Partial Order</td>
<td>WR-RW Partial order</td>
</tr>
<tr>
<td>Multiple Item</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I) Per</td>
<td>RWx Partial Order</td>
<td>RW-WWx Partial Order</td>
</tr>
<tr>
<td>Single Item</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Requiring, as we do in the **not-read-before-write case**,

\[
\text{WR-RW}(H) \text{ Partial Order} \quad \text{[Global]}
\]

\&

\[
\forall x \in D \left[ [\text{WR-}]RW-WW_x(H) \text{ Partial Order} \right] \quad \text{[Local]}
\]

means that for each item where there is some WR- and/or RW-relationship(s), the corresponding local serialization order part must be consistent with the global serialization order. But for each item where there is no WR- and/or RW-relationship(s), the corresponding local serialization order may be non-consistent with the global serialization order.

(Remember that a serialization order is only unique when the corresponding binary relation is a total order, while it is non-unique when the corresponding binary relation is only a partial order).

We effectively achieve in both the read-before-write and the not-read-before-write cases:

- m possibly different **local serialization orders**

\[
(m = |D| - \text{ being the number of items})
\]

For each separate item - \( x \in D \) - this is a maxi-order. This means that it corresponds to all types of conflicts; i.e. we have a

\[
\text{WR-RW-WW}_x \text{ order}.
\]

Hence we do implement atomic actions per single item (like in the operating systems model in Section 2.1.2).
Some of the local serialization orders may be mutually consistent, and some may be mutually inconsistent. Hence we do not implement atomic transactions per multiple item (unlike in the database systems model still in Section 2.1.2).

- 1 necessarily common global serialization order

For all items together this is a mini-order. This means that it does not correspond to all types of conflicts; i.e. we have a

WR / WR-RW order.

Some of the local serialization order parts (those with WR- / WR-RW-relationship(s)) must be consistent with the global serialization order, while some others (those without WR- / WR-RW-relationship(s)) may be non-consistent with the global serialization order.

Paraphrased, we implement in both the read-before-write and not-read-before-write cases:

- common

WR-RW-WW, synchronization \([x \in D]\)

per single item separately

(mutually consistent or non-consistent),

and

- common

WR / WR-RW synchronization

per multiple item collectively

(consistent with the ones above where applicable),

but

- non-common

(WR-)RW-WW / (WR-RW-)WW synchronization

per multiple item collectively

(our gained flexibility).
Let us now refer back to Figs. 9.13 and 9.14.

First, we will look at the read-before-write case.

Going from totally requiring

WR-RW-WW(Any Schedule) Partial Order

to globally requiring

WR(Any Schedule) Partial Order

allows deleting RW-arcs and/or WW-arcs from the corresponding serialization graph.

Actually, deleting only WW-arcs cannot bring us from a cyclic graph to an acyclic graph because of the read-before-write restriction.

But, deleting both WW-arcs and RW-arcs may bring us from a cyclic graph to an acyclic graph. For this type of deletion only the subcycle version of Fig. 9.13 applies because of the read-before-write restriction and the specific local criterion.

Further, deleting only RW-arcs may bring us from a cyclic graph to an acyclic graph. For this type of deletion both the subcycle version of Fig. 9.13 and the total cycle version of Fig. 9.14 apply.

(For an in-depth treatment, see Sections 9.3.1 and 9.4.2).

Second, we will look at the not-read-before-write case.

Going from totally requiring

WR-RW-WW(Any Schedule) Partial Order

to globally requiring

WR-RW(Any Schedule) Partial Order

allows deleting WW-arcs from the corresponding serialization graph.

Here, deleting only WW-arcs may bring us from a cyclic graph to an acyclic graph because of the not-read-before-write freedom. For this type of deletion both the subcycle version of Fig. 9.13 and the total cycle version of Fig. 9.14 apply.

(For an in-depth treatment, see Sections 9.3.2 and 9.4.3).
Let us then combine this with our above statements concerning serialization order consistency or not.

Actually, in the case of a subcycle deletion, i.e. as exemplified in Fig. 9.13, the resulting global serialization order must be non-consistent with at least one of the resulting local serialization orders. Further all the local serialization orders may not be mutually consistent - as we are assuming the non-existence of a total serialization order.

However, in the case of a total cycle deletion, i.e. as exemplified in Fig. 9.14, the resulting global serialization order may be consistent with all of the resulting local serialization orders. Still all the local serialization orders may not be mutually consistent - as we are assuming the non-existence of a total serialization order.

(Remember again that a serialization order is non-unique when the corresponding binary relation is only a partial order, while it is only unique when the corresponding binary relation is a total partial order).

With respect to illustrations of the effects stated here concerning both subcycle deletion and total cycle deletion, see the discussions of the examples to come in the rest of this chapter - and especially those in Sections 9.3.1 and 9.3.2.

To round off, in our systems we do not allow a local decision (based on one retrieval of a single item) of a specific transaction to be invalidated (with respect to a later update of this item) by other transactions. But we do allow global decisions (based on several retrievals of multiple items) of a specific transaction to be influenced (with respect to some later updates of such items) by other transactions.

In which ways this influence may have effects will be different among the read-before-write and not-read-before-write cases, and the two different cases will be treated in Sections 9.3.1 and 9.3.2 respectively.
9.2.2 Special Analysis and Discussions

Now we shall carry out some specific investigations needed in the discussions of the classes $A^R_{SR}$ and $A^N_{NR}SR$ in Section 9.2.1.

Local/Global-Relations Coherence

From the existence of a specific pair of conflicting operations in a system schedule - see Eq. 2.6 in Section 2.2.4, what may additionally be concluded with respect to the existence of other specific pair(s) of conflicting operations?

First, let us imagine the following write-write pattern occurring in a schedule $H$:

\[-W_i(w) \rightarrow W_j(w) \quad (Eq. 9.67)\]

For the read-before-write case (as $R_i(w) < W_i(w)$ always) there must also be a corresponding read-write pattern in the schedule $H$:

\[\rightarrow R_i(w) \rightarrow W_j(w)\]

But for the not-read-before-write case there must not also be any corresponding read-write pattern in the schedule $H$:

\[\rightarrow \quad \text{---------}\]

Second, let us imagine the following write-read pattern occurring in a schedule $H'$:

\[-W_i(w) \rightarrow R_j(w) \quad (Eq. 9.68)\]

For the read-before-write case (as $R_i(w) < W_i(w)$ always - and $R_j(w) < W_j(w)$ sometimes) there must - on the condition that $W_i(w)$ does occur in $H'$ - also be both a corresponding read-write pattern and a corresponding write-write pattern in the schedule $H'$:

\[\rightarrow R_i(w) \rightarrow W_j(w) \quad \& \quad W_i(w) \rightarrow W_j(w) \quad \text{Given That } W_j(w) \text{ Exists}\]

Though for the not-read-before-write case (as $R_j(w) < W_i(w)$ sometimes) there must - on the condition that $W_i(w)$ does occur in $H'$ - also be only a corresponding write-write pattern in the schedule $H'$:

\[\rightarrow \quad \text{---------} \quad \& \quad W_i(w) \rightarrow W_j(w) \quad \text{Given That } W_j(w) \text{ Exists}\]
Third, let us imagine the following \textit{read-write} pattern occurring in a schedule $H''$:

$$ R_i(w) \rightarrow W_j(w) \quad \text{(Eq. 9.69)} $$

For the \textit{read-before-write case} there must not also be a corresponding pattern of any kind in the schedule $H''$ - even on the condition that $W_i(w)$ does occur in $H''$:

$$ \rightarrow \text{----------------} \quad \text{Even Given That} \; W_i(w) \text{ Exists} $$

And for the \textit{not-read-before-write case} there must not also be a corresponding pattern of any kind in the schedule $H''$ - even on the condition that $W_i(w)$ does occur in $H''$:

$$ \rightarrow \text{----------------} \quad \text{Even Given That} \; W_i(w) \text{ Exists} $$

To summarize, only from the existence of a \textit{write-write} pair of conflicting operations in the \textit{read-before-write case} is it possible unconditionally to conclude something additional with respect to the existence of other specific pair(s) of conflicting operations in a system schedule.
LOCAL-CRITERIA COHERENCE

With respect to the definition(s) of x-conflict equivalent schedules and x-conflict serializable schedules - see Eqs. 9.8 to 9.10 in Section 9.1.1, does the binary relation WR-RW-WWₙ(Any Schedule) - x ∈ D - really have to be a partial order?

First, let us look at all possible combinations of accesses to a single item x from two transactions T₁ and T₂ only.

Here the read-before-write case may be split into three groups of a total of 11 different cases.

For the two transactions both reading and writing the item, we have the following six possibilities:

\[ H_{51} = \begin{cases} T_1: R_1(x)W_1(x) \\ T_2: R_2(x)W_2(x) \end{cases} \]

\[ H_{52} = \begin{cases} T_1: R_1(x)W_1(x) \\ T_2: R_2(x)W_2(x) \end{cases} \]

\[ H_{53} = \begin{cases} T_1: R_1(x)W_1(x) \\ T_2: R_2(x)W_2(x) \end{cases} \]

\[ H_{54} = \begin{cases} T_1: R_1(x)W_1(x) \\ T_2: R_2(x)W_2(x) \end{cases} \]

\[ H_{55} = \begin{cases} T_1: R_1(x)W_1(x) \\ T_2: R_2(x)W_2(x) \end{cases} \]

\[ H_{56} = \begin{cases} T_1: R_1(x)W_1(x) \\ T_2: R_2(x)W_2(x) \end{cases} \]

We see that \( H_{51} \) and \( H_{56} \) are serial (and symmetric) schedules. Further we see that \( H_{52} \) and \( H_{55} \) are symmetric schedules, and also that \( H_{53} \) and \( H_{54} \) are symmetric schedules.

The CSGₙ of these six schedules are given in Figs. 9.17a to 9.17f.

From the above figures we may deduce that \( H_{52} \) and \( H_{54} \) are x-conflict equivalent schedules, and also that \( H_{53} \) and \( H_{55} \) are x-conflict equivalent schedules. But we must conclude that only \( H_{51} \) and \( H_{56} \) are x-conflict serializable schedules.
Fig. 9.17a. CSGₜ(Hₜ).

Fig. 9.17b. CSGₜ(Hₚ).

Fig. 9.17c. CSGₜ(Hₚ).

Fig. 9.17d. CSGₜ(Hₚ).
For the one transaction only reading the item plus the other transaction both reading and writing the item, we have the following three possibilities:

\[
H_{57} = \\
T_1: R_1(x) \\
T_2: R_2(x)W_2(x)
\]

\[
H_{58} = \\
T_1: R_1(x) \\
T_2: R_2(x) \quad W_2(x)
\]

\[
H_{59} = \\
T_1: R_1(x) \\
T_2: R_2(x)W_2(x)
\]

We see that \(H_{57}\) and \(H_{59}\) are serial schedules.

The CSGs of these three schedules are given in Figs. 9.17g to 9.17i.

From the above figures we may deduce that \(H_{57}\) and \(H_{58}\) are x-conflict equivalent schedules. So we may conclude that \(H_{57}, H_{58}\) and \(H_{59}\) are naturally all x-conflict serializable schedules.
For the two transactions only reading the item, we have the following two possibilities:

\[
H_{60} = \begin{align*}
& T_1: R_1(x) \\
& T_2: R_2(x)
\end{align*}
\]

\[
H_{61} = \begin{align*}
& T_1: R_1(x) \\
& T_2: R_2(x)
\end{align*}
\]

We see that both \(H_{60}\) and \(H_{61}\) are serial (and symmetric) schedules.

The CSGs of these two schedules are given in Figs. 9.17j to 9.17k.
So we may conclude that $H_{60}$ and $H_{61}$ are naturally both $x$-conflict serializable schedules.

Thus having exhausted all the possibilities ($H_{51}$ to $H_{61}$) for the read-before-write case, the CSG's show that all the non-$x$-conflict serializable schedules ($H_{52}$ to $H_{55}$) correspond to the binary relation $RW_{xx}(The\ Schedule)$ being a non-partial order. Actually even the binary relation $RW_x(The\ Schedule)$ constitutes a non-partial order in each case.

Further each of the non-$x$-conflict serializable schedules effectively represents a lost update problem. Hence this seems to be the only problem which possibly occurs in the read-before-write case per single item (for two transactions only).

Now the not-read-before-write case may be split into six groups of a total of 18 different cases.

The first three groups of a total of 11 different cases are the same as those covered above. Thus informally we have:

$$H_{51} = H_{51}$$
.. 
.. 

$$H_{61} = H_{61}$$
The last three groups of a total of seven different cases are new and will accordingly be covered below.

For the one transaction only writing the item plus the other transaction both reading and writing the item, we have the following three possibilities:

\[ H_{62} \]
\[
\begin{align*}
T_1 &: W_1(x) \\
T_2 &: R_2(x)W_2(x)
\end{align*}
\]

\[ H_{63} \]
\[
\begin{align*}
T_1 &: W_1(x) \\
T_2 &: R_2(x) W_2(x)
\end{align*}
\]

\[ H_{64} \]
\[
\begin{align*}
T_1 &: W_1(x) \\
T_2 &: R_2(x)W_2(x)
\end{align*}
\]

We see that \( H_{62} \) and \( H_{64} \) are serial schedules.

The CSGs of these three schedules are given in Figs. 9.18a to 9.18c.
From the above figures we must conclude that only $H_{62}'$ and $H_{64}'$ are $x$-conflict serializable schedules.

For the one transaction only writing the item plus the other transaction only reading the item, we have the following two possibilities:

\[
H_{65}' = \\
T_1: W_1(x) \\
T_2: R_2(x)
\]

\[
H_{66}' = \\
T_1: W_1(x) \\
T_2: R_2(x)
\]

We see that both $H_{65}'$ and $H_{66}'$ are serial schedules.

The $\text{CSG}_x$s of these two schedules are given in Figs. 9.18d to 9.18e.
So we may conclude that $H_{65}'$ and $H_{66}'$ are naturally both $x$-conflict serializable schedules.

For the two transactions only writing the item, we have the following two possibilities:

$H_{67}' =
\begin{align*}
T_1 & : W_1(x) \\
T_2 & : W_2(x)
\end{align*}$

$H_{68}' =
\begin{align*}
T_1 & : W_1(x) \\
T_2 & : W_2(x)
\end{align*}$

We see that both $H_{67}'$ and $H_{68}'$ are serial (and symmetric) schedules. The CSG's of these two schedules are given in Figs. 9.18f to 9.18g.

![Fig. 9.18f. CSG(H_{67}').](image)

![Fig. 9.18g. CSG(H_{68}').](image)

So we may conclude that $H_{67}'$ and $H_{68}'$ are naturally both $x$-conflict serializable schedules.

Thus having also exhausted all the possibilities ($H_{51}'$ to $H_{68}'$) for the not-read-before-write case, the CSG's show that all the non-$x$-conflict serializable schedules ($H_{52}'$ to $H_{65}'$ and $H_{63}'$) correspond to the binary relation $RW-WW_x$ (The Schedule) being a non-partial order. (But compared to the read-before-write case conclusion above, it is now not a valid conclusion that even the binary relation $RW_x$ (The...
Schedule) constitutes a non-partial order in each case - because of the extra $H_{63}$ possibility).

Further again each of the non-x-conflict serializable schedules effectively represents a lost update problem. Hence this also seems to be the only problem which possibly occurs in the not-read-before-write case per single item (for two transactions only).

Second, let us imagine having more than two transactions accessing a single item.

It is easy to see that a conflict between several transactions may always be separated into a set of conflicts between two and two transactions. Thus we will have at least the same relationships in the binary relation WR-RW-WW, (Any Schedule) and correspondingly at least the same arcs in the CSG, (The Schedule). We may have more contributions, but never less. (This may be shown formally by an induction-proof).

From this we may deduce that any non-x-conflict serializable schedule $H$ (i.e. representing at least one lost update problem) will always correspond to at least the binary relation $RW[-WW]_x(H)$ being a non-partial order in the read-before-write case - and at least the binary relation $RW-WW_x(H)$ being a non-partial order in the not-read-before-write case.

To summarize, to make any schedule $H$ at least x-conflict serializable in the read-before-write case, it is enough only to assure

$$RW_x(H) \text{ Partial Order } [x \in D].$$

Likewise, to make any schedule $H$ at least x-conflict serializable in the not-read-before-write case, it is enough only to assure

$$RW-WW_x(H) \text{ Partial Order } [x \in D].$$

We are actually not saying that this is necessary to avoid lost update problems, only that it is sufficient).

Note that the lack of a WR-part in both cases even follows fairly intuitively. It stems indirectly from the fact in Section 9.1.2 stating that the binary relation $WR_x$(Any Schedule) - $x \in D$ - is a partial order per definition. (See also Eq. 9.68 in the previous paragraph).

Observe further that the difference between the results for the two cases, i.e. the lack of even the WW-part in the read-before-write case, also follows from the results of the previous paragraph. Remember that any write-write relationship in a binary relation [WR] $RW-WW$(The Schedule) or [WR]RW-WW, (The Schedule) - $x \in D$ - will be mirrored by a corresponding read-write relationship in the read-before-write case (see Eq. 9.67).
MODEL EFFECTS

With respect to the specifications of correctness criteria for the
read-before-write and not-read-before-write cases — see respectively
Eqs. 9.44 and 9.48 in Section 9.2.1, are the correctness criteria
applicable both to the general multi-action-single-item model in
Chapter 2 and the special single-action-multi-item model in Chapter 4?

First, let us look at the class \( A_R \)SR for the read-before-write case.

The correctness criterion part per multiple item corresponds to
assuring that the binary relation \( WR(H) \) is a partial order for any
schedule H. Thus the aim is to avoid cycles in the \( A_R \)SG(H) of the type
given in Fig. 9.19.

It is easy to see that cycles like the one shown above may never occur
in the single-action-multi-item model. This stems once more from a
fact similar to the one in Section 9.1.2 stating that the binary
relation \( WR_X(\text{Any Schedule}) - x \in D \) is a partial order per
definition. In the single-action-multi-item model even the binary
relation \( WR(\text{Any Schedule}) \) is a partial order (both for the read-
before-write and not-read-before-write cases of course). Remember from
Section 4.1 both the effective restriction of maximum one read-action
and/or one write-action per transaction and the effective prohibition
against a read-action to succeed a write-action in order in a
transaction.

Thus the given correctness criterion is not applicable for this model.
Or paraphrased; the given correctness criterion is always fulfilled —
i.e. the global part of the membership requirements for class \( A_R \)SR is
assured for any schedule without any synchronization in the single-
action-multi-item model.

But for the multi-action-single-item model the given correctness
criterion is real — in the sense that cycles like the one shown above
may occur if no synchronization is applied.

An example of a schedule resulting in a cycle of the discussed kind,
is the following:
\[ H_{69} = \]
\[ T_1: R_1(x) W_1(x) \]
\[ T_2: R_2(x) R_2(y) W_2(y) \]

The \( A_{SG} \) of this schedule is given in Fig. 9.19a.

![Fig. 9.19a. CSG(H_{69}).](image)

Second, let us look at the class \( A_{NR-SR} \) for the not-read-before-write case.

The correctness criterion part per multiple item corresponds to assuring that the binary relation WR-RW(H) is a partial order for any schedule \( H \). Thus the aim is to avoid cycles in the \( A_{NR-SG}(H) \) of the type given in Fig. 9.20.

![Fig. 9.20. A Cycle corresponding to both WR- and RW-relationships.](image)

Now both for the single-action-multi-item model and the multi-action-single-item model the given correctness criterion is real - in the sense that cycles like the one shown above may occur in both models if no synchronization is applied.

The simplest example of a schedule in the single-action-multi-item model resulting in a cycle of the discussed kind, is the following:

\[ H_{69} = \]
\[ T_1: R_1(x) W_1(x) \]
\[ T_2: R_2(x) W_2(x) \]
The "\(A_{NR SG}\)" of this schedule is given in Fig. 9.20a.

![Fig. 9.20a. CSG\((H_{69}))\].

Thus the given schedule actually corresponds to a cycle representing only RW-relationships. Even so, it complies with the read-before-write requirements and contains a per single item problem - i.e. schedule \(H_{69}''\) equals schedule \(H_3\) in Section 9.1.2 representing a lost update situation.

Let us also show an example of a schedule - resulting in a cycle of the discussed kind in the single-action-multi-item model - that corresponds to a cycle representing both WR- and RW-relationships. Further, it does not comply with the read-before-write requirements and does contain a per multiple item problem:

\[
H_{69}'' =
\begin{align*}
T_1 &: R_1(x) \\
T_2 &: W_2(x,y) \\
T_3 &: R_3(y,z)
\end{align*}
\]

The \(A_{NR SG}\) of this schedule is given in Fig. 9.20b.

![Fig. 9.20b. CSG\((H_{69}''\))].

As the multi-action-single-item model is a generalization of the single-action-multi-item model, actually both \(H_{69}''\) and \(H_{69}'''\) (and even \(H_{69}\)) are valid examples of problematic schedules in the multi-action-single-item model too (for the not-read-before-write case).
To summarize, only for the combination of the single-action-multi-item model with the read-before-write case is our correctness criterion less relevant. This means that the global part of the membership requirements for class $A_{pSR}$ is assured for any schedule without any synchronization in the single-action-multi-item model.
CRITERIA BASIS

Observing the existence of a multitude of general correctness criteria like for example FSR, VSR₀, VSRₙ and CSR from Chapters 2 and 4, why settle for the least general CSR as a starting point for our new correctness criteria?

This question arises when one notices that as well the local criteria
\[
RWᵪ(H) / RW-WWᵪ(H) \text{ Partial Order } [x \in D]
\]
as the global criteria
\[
WR(H) / WR-RW(H) \text{ Partial Order}
\]
corresponding to \( H \in AᵣSR / AᵣₙSR \) are in both the read-before-write and not-read-before-write cases subversions of the total criterion
\[
WR-RW-WW(H) \text{ Partial Order}
\]
corresponding to \( H \in CSR \).

(Remember that \( H \in CMₓ, x \in D \) - corresponds to \( H[x] \in CSR \)).

First, let us look at the read-before-write case.

From Eq. 4.52 in Section 4.3 we have for the special single-action-multip-item model in Chapter 4:

\[
FSR = VSR₀ = VSRₙ = CSR
\]

The local access per single item must necessarily correspond to a further specialization of this model - i.e. a "single-action-single-item" model allowing maximum one read and one write per item per transaction (see also Section 2.2.4).

Thus CSR represents the one (common) possible starting point for the local criterion.

From Eq. 4.51 in Section 4.3 we have for the general multi-action-single-item model in Chapter 2:

\[
FSR > VSR₀ = VSRₙ = CSR
\]

The global access per multiple item must necessarily correspond exactly to this model.

Thus - short of FSR - CSR again represents the one (common) possible starting point even for the global criterion. (And to have easily combinable criteria locally and globally, we settle by not investigating the possibility of basing the global criterion on FSR. See also Section 9.4.1).
Second, let us look at the not-read-before-write case.

From Eq. 4.50 in Section 4.3 we have both for the special single-action-multi-item model in Chapter 4 and the general multi-action-single-item model in Chapter 2:

\[ \text{FSR} \supset VSR_0 \supset VSR_N \supset \text{CSR} \]

Thus, CSR does not represent the only possible starting point neither for the local criterion nor the global criterion. But to have easily comparable criteria for the read-before-write and not-read-before-write cases, we settle by not investigating the possibility of basing either the local criterion or the global criterion on any other classes than CSR.

To summarize, CSR is formally not the only possible starting point for our new correctness criteria in all cases. But to have combinable and comparable criteria, CSR is the natural choice.

(Practically we also have to take the efficiency aspects into consideration. As mentioned in Section 4.3 this may rule out anything but CSR as criteria basis even for the remaining cases where there still are theoretically other possibilities).
CASE GROUPING

Having carried out a parallel treatment of the read-before-write and not-read-before-write cases, are both really necessary and/or interesting?

First, the motive for covering the read-before-write case is that this case corresponds directly and naturally to the alternative database and transaction types described in Sections 6.2 and 7.2.

Thus the reason for dealing with this case is pure necessity. Our alternative systems demand such a basic analysis.

Second, the motive for also covering the not-read-before-write case is that this case opens the way for a natural and direct generalization/extension of our alternative specifications.

Thus the reason for dealing with this case is pure interest. Our alternative systems allow such an extra possibility.

To summarize, our main emphasis will still be on the special read-before-write case and its corresponding correctness criterion. But we will also continue to put some emphasis on an accompanied treatment of the general not-read-before-write case and its possible correctness criterion. And we will do this even without yet being able to state the exact application area of the results or the eventual combination of the results with those of the main case in its corresponding application area; i.e. wander-transactions in a skeleton-database.
9.3 Resulting Freedom and Constraints

In this section we will illustrate the immediate consequences of our new specifications. Thus we shall show what kind of situations or schedules are allowed and what kind of situations or schedules are not allowed for the two separate cases of read-before-write and not-read-before-write.

9.3.1 Initial Characterization of Read-before-write Class

Here we start by investigating four different situations for the read-before-write case. Two of these situations (A and B) will be allowed according to the definition of ARSR, and the other two (A' and B') will not.

**Situation A**

First, let us imagine the following two patterns occurring together in a schedule $H_7$:

$$R_1(x) \rightarrow R_2(x)W_2(x)$$

$$\&$$

$$R_2(y)W_2(y) \rightarrow R_1(y)[W_1(y)]$$

[\[x \neq y\]]

(With respect to the meaning of such patterns, refer to the comments concerning Fig. 2.13 in Section 2.2.4. Further, the actions inside squares are those where special attention is to be paid, while the bracketed action(s) may either be included or excluded without changing the main characteristics of a given situation).

We see that the $H_7$-situation - excluding the bracketed $W_1(y)$ - corresponds to schedule $H_4$ from Section 9.1.2. Thus this a (single) inconsistent retrieval situation. One transaction $T_1$ sees a part, $y$, of another transaction $T_2$'s result when retrieving the corresponding item (and later updating it), without $T_1$ seeing the rest, $x$, of $T_2$'s result when retrieving that corresponding item.

From a cyclic graph in Fig. 9.21a (or 9.21c) we have that

$$H_7 \neq_c \text{ Any Permutation of } T_1 \& T_2,$$

and thus we must deduce that:

$$H_7 \notin \text{ CSR}$$

But from Fig. 9.22 we see that:

$$H_7 =_{ra} T_2 \otimes T_1$$
And from Figs. 9.22a and 9.22b (or 9.22c and 9.22d) we see that:

$$H_7 =_{x_c} T_1 \otimes T_2$$
$$H_7 =_{y_c} T_2 \otimes T_1$$

Then we may conclude that:

$$H_7 \in A_R SR$$

So a single inconsistent retrieval situation is allowed in the read-before-write case. Even if we include an extra update like $W_1(y)$ in such a situation, the validity of this fact does not change.
For \( H_1 \), excluding \( W_1(y) \) the difference between the corresponding CSG and the corresponding \( A^SG \) is only RW-type arc(s), while for \( H_2 \), including \( W_1(y) \) the difference is both RW-type and WW-type arcs. In both these variants we only delete one (or more) arc(s) in a cycle.
and not all arcs - going from CSG to $A_{R}SG$. (In the second variant we do delete arc-sources constituting a complete cycle, but because of duplicate arc-sources we do not delete a total cycle - only a subcycle). This corresponds to the subcase from Fig. 9.13 in Section 9.2.1.
SITUATION A‘

Now, let us add an update \( W_1(x) \) to the above situation A. After also bracketing the existing update \( W_2(x) \), we end up with the following two patterns occurring together in a schedule \( H_{7'} \):

\[
R_1(x)W_1(x) - [R_2(x)] [W_2(x)]
\]

\[
&
R_2(y)W_2(y) - [R_1(y)] [W_1(y)]
\]

We see that the \( H_{7'} \)-situation - including the bracketed \( W_1(y) \) and \( W_2(x) \) - corresponds to schedule \( H_1 \) from Section 9.1.1. This might be called a double inconsistent retrievals situation. A first transaction \( T_1 \) sees (one part of, i.e. \( y \)) a second transaction \( T_2 \)'s result when retrieving the corresponding item (and later updating it), while \( T_2 \) again sees (another part of, i.e. \( x \)) \( T_1 \)'s result when retrieving that corresponding item (and later updating it).

From a cyclic graph in Fig. 9.23a (or 9.23c) we have that

\( H_{7'} \neq \text{Any Permutation of } T_1 \& T_2, \)

and thus we must deduce that:

\( H_{7'} \in \text{CSR} \)

---

Fig. 9.23a. CSG(\( H_{7'} \) incl. \( W_1(x) \) & \( W_1(y) \)).

---

Fig. 9.23c. CSG(\( H_{7'} \) excl. \( W_2(x) \) & \( W_1(y) \)).
Here from a cyclic graph in Fig. 9.24 we further see that:

* $H_7' \notin A_{ra}$ Any Permutation of $T_1$ & $T_2$

![Fig. 9.24. $A_{ra}SG(H_7')$.](image)

And from Figs. 9.24a and 9.24b (or 9.24c and 9.24d) we see that:

\[ H_7' =_{xc} T_1 \circ T_2 \]
\[ H_7' =_{yc} T_2 \circ T_1 \]

![Fig. 9.24a. $CSG(H_7' \text{ incl. } W_2(x) \& W_1(y))$.](image)

![Fig. 9.24b. $CSG(H_7' \text{ incl. } W_2(x) \& W_1(y))$.](image)

So (from the starred fact) we must now even conclude that:

\[ H_7' \notin A_{SR} \]

Thus a double inconsistent retrievals situation is not allowed in the read-before-write case. Even if we exclude extra updates like $W_1(y)$ and $W_2(x)$ from such a situation, the validity of this fact does not
change. When such mutually inconsistent retrieving with two transactions and two items is generalized into several transactions and several items, we call it a recursive inconsistent retrievals situation. This is still not allowed.

The difference between the allowed situation A and the non-allowed situation A' may be described as follows. A illustrates a situation where one transaction has taken a decision with regard to an item upon whose result another transaction may base its decision(s), while A' illustrates a situation where two (or more) transactions have taken decisions with regard to different items upon whose results they both (or all) recursively should base their further decisions. Thus it is allowed for one transaction to see only partial - and not total - results of another transaction, but it is not allowed for two (or more) transactions to see only partial results of each other recursively.
SITUATION B

Second, let us delete the update \( W_2(y) \) from the original situation A. After also unbracketing the existing update \( W_1(y) \), we end up with the following two patterns occurring together in a schedule \( H_8 \):

\[
\begin{align*}
R_1(x) & \rightarrow R_2(x) W_2(x) \\
\& \\
R_2(y) & \rightarrow R_1(y) W_1(y)
\end{align*}
\]

\([x \neq y]\)

This variant or part of the \( H_2 \)-situation might also be called a **double dependent updates situation**. A first transaction \( T_1 \) sees (one part of, i.e. \( x \)) the basis for a second transaction \( T_2 \)'s decision with regard to an item by retrieving this item before it is updated, while \( T_2 \) again sees (another part of, i.e. \( y \)) the basis for \( T_1 \)'s decision with regard to an item by retrieving that item before it is updated.

From a cyclic graph in Fig. 9.25 we have that

\[ H_8 \not\in_c \text{Any Permutation of } T_1 \& T_2, \]

and thus we must deduce that:

\[ H_8 \not\in \text{CSR} \]

![Fig. 9.25. CSG(H_8).](image-url)

But from Fig. 9.26 we see that:

\[ H_8 \not\in^a T_1 \circ T_2 =^a T_2 \circ T_1 \]

And from Figs. 9.26a and 9.26b we see that:

\[ H_8 \not\in^c T_1 \circ T_2 \]

\[ H_8 \not\in^c T_2 \circ T_1 \]

Then we may conclude that:

\[ H_8 \in A_8 SR \]
So a double dependent updates situation is allowed in the read-before-write case. When such mutually dependent updating with two transactions and two items is generalized into several transactions and several items, we call it a recursive dependent updates situation. This is still allowed.

For $H_8$ the difference between the corresponding CSG and the corresponding $A_8$SG is only RW-type arcs. In this variant we delete all arcs in a cycle going from CSG to $A_8$SG. This corresponds to the subcase from Fig. 9.14 in Section 9.2.1. (A valid example of this subcase combined with deleting both RW-type and WW-type arcs in a cycle is impossible here because of the read-before-write restriction and the specific local criterion).
SITUATION B′

Now, let us equalize the items x and y in the above situation B. We end up with the following two patterns occurring together in a schedule $H_{y}$:

$$
R_1(x) \rightarrow W_2(x) \\
\& \\
R_2(x) \rightarrow W_1(x)
$$

We see that the $H_{y}$'-situation - including also a $W_1(x)$-$W_2(x)$ ordering, and observing of course the $R_1(x) \prec W_1(x)$ and $R_2(x) \prec W_2(x)$ requirements - corresponds to schedule $H_3$ from Section 9.1.2. Thus this a (single) lost update situation. Two transactions $T_1$ and $T_2$ both see (part of, i.e. x) the basis for both transactions' decisions with regard to an item by both retrieving this item before it is updated by any of them.

From a cyclic graph in Fig. 9.27 we have that

$H_{y} \neq_c$ Any Permutation of $T_1$ & $T_2$,

and thus we must deduce that:

$H_{y} \notin CSR$

Here from Fig. 9.28 we see that:

$H_{y} =_r T_1 \bowtie T_2 =_r T_2 \bowtie T_1$

And from a cyclic graph in Fig. 9.28a we further see that:

* $H_{y} \neq_x c$ Any Permutation of $T_1$ & $T_2$

So (from the starred fact) we must now even conclude that:

$H_{y} \notin A_r SR$

Thus a (single) lost update situation is (naturally) not allowed in the read-before-write case.
The difference between the allowed situation B and the non-allowed situation B' may be described as follows. B illustrates a situation where two (or more) transactions may take decisions with regard to separate different items whose bases both (or all) recursively have seen, while B' illustrates a situation where two transactions should take decisions with regard to a unique item whose basis both have seen.
9.3.2 Initial Characterization of Not-read-before-write Class

Now we investigate four different situations for the not-read-before-write case. Again, two of these situations (C and D) will be allowed according to the definition of $A_{NR}$, and the other two ($C'$ and $D'$) will not.

(As mentioned in Section 9.2.1 our treatment of the not-read-before-write case will not be as complete and general as for the read-before-write case. Thus the situations to be given do not represent an exhaustive exploration of the possibilities, but more a basic illustration of the properties).

**Situation C**

First, let us imagine the following three patterns occurring together in a schedule $H_g$:

\[
\begin{align*}
R_1(x) & \quad [[W_1(x)]] - W_2(x) \\
& \quad \& \\
W_2(z) & \quad - W_3(z) \\
& \quad \& \\
W_3(y) & \quad - R_1(y) \quad [W_1(y)]
\end{align*}
\]

This might be called an *indirect inconsistent retrieval situation*. Transaction $T_1$ sees a part, $y$, of transaction $T_3$'s result when retrieving the corresponding item (and later updating it), but transaction $T_1$ does not see a part, $x$, of transaction $T_2$'s result when retrieving that corresponding item (and later updating it) - even though transactions $T_2$ and $T_3$ have a common result-part, $z$, whose corresponding updates necessarily have to be ordered.

From a cyclic graph in Fig. 9.29a (or 9.29d) we have that

\[ H_g \neq_c \text{Any Permutation of } T_1, T_2 \& T_3, \]

and thus we must deduce that:

\[ H_g \notin CSR \]

But from Fig. 9.30 we see that:

\[ H_g =_{nra} T_3 \otimes T_1 \otimes T_2 \]
And from Figs. 9.30a, 9.30b and 9.30c (or 9.30d, 9.30e and 9.30f) we see that:

\[ H_9[x] =_c T_1[x] \circ T_2[x] \]
\[ H_9[y] =_c T_3[y] \circ T_1[y] \]
\[ H_9[z] =_c T_2[z] \circ T_3[z] \]

Then we may conclude that:

\[ H_9 \in \mathbb{A}_{NRSR} \]
So an indirect inconsistent retrieval situation is allowed in the not-read-before-write case. Even if we include extra updates like $W_1(y)$ and $W_1(x)$ in such a situation, the validity of this fact does not change.

For $H_9$, either excluding or including $W_1(y)$ and $W_1(x)$ the difference between the corresponding CSG and the corresponding $A_{NR-SG}$ is (per definition) only WW-type arc(s). In both these variants we only delete one (or more) arc(s) in a cycle - and not all arcs - going from CSG to $A_{NR-SG}$. (Once more in the second variant we do delete arc-sources constituting a complete cycle, but because of duplicate arc-sources we do not delete a total cycle - only a subcycle). Again this corresponds to the subcase from Fig. 9.13 in Section 9.2.1.
Fig. 9.30d. $\text{CSG}_i[H_3 \text{ incl. } W_i(y) \& W_i(x)]$.

Fig. 9.30e. $\text{CSG}_i[H_3 \text{ incl. } W_i(y) \& W_i(x)]$.

Fig. 9.30f. $\text{CSG}_i[H_3 \text{ incl. } W_i(y) \& W_i(x)]$. 
SITUATION C'

Now, let us equalize the items \( z \) and \( y \) (or alternatively \( x \) and \( z \)) in the above situation C. After also excluding the double-bracketed \( W_1(x) \), we end up with the following two patterns occurring together in a schedule \( H_9' \):

\[
\begin{align*}
{R_1(x)} & \quad - \quad \{W_2(x) \\
\& \quad W_2(y) - \quad \{R_1(y) \quad [W_1(y)]
\end{align*}
\]

We see that the \( H_9' \)-situation mirrors the \( H_7 \)-situation [see A] - only excluding the retrievals \( R_2(x) \) and \( R_2(y) \), and hence the \( H_9' \)-situation corresponds to schedule \( H_4' \) from Section 9.1.2. Thus this is once more the (single) inconsistent retrieval situation.

From a cyclic graph in Fig. 9.31a (or 9.31c) we have that

\( H_9' \neq CSR \)

\[
\begin{align*}
\text{Fig. 9.31a. CSG(H_9', incl.W(y)).}
\end{align*}
\]

\[
\begin{align*}
\text{Fig. 9.31c. CSG(H_9', excl.W(y)).}
\end{align*}
\]

Here from a cyclic graph in Fig. 9.32 we further see that:

\( \star \) \( H_9' \neq_{nra} \) Any Permutation of \( T_1 \) & \( T_2 \)
And from Figs. 9.32a and 9.32b (or 9.32c and 9.32d) we see that:

\[ H \gamma' =_{x_c} T_1 \sqcap T_2 \]
\[ H \gamma' =_{y_c} T_2 \sqcap T_1 \]

So (from the starred fact) we must now even conclude that:

\[ H \gamma' \notin A_{NRSR} \]

Thus a single inconsistent retrieval situation is not allowed in the not-read-before-write case. This is contrary to in the read-before-write-case (see situation A). Even if we exclude an extra update like \( W_1(y) \) from such a situation, the validity of this fact does not change.
And the same applies of course also to the more general double or even recursive inconsistent retrievals situation (see situation A'). We may call all these non-allowed situations - single inconsistent retrieval and double or recursive inconsistent retrievals - direct inconsistent retrieval(s) situations.
SITUATION D

Second, let us delete the retrievals \( R_1(x) \) and \( R_1(y) \) and equalize the items \( z \) and \( y \) (or alternatively \( x \) and \( z \)) in the original situation C. After also unbracketing the existing updates \( W_1(y) \) and \( W_1(x) \), we end up with the following two patterns occurring together in a schedule \( H_{10} \):

\[
\begin{align*}
W_1(x) & \rightarrow W_2(x) \\
\& & [x \neq y] \\
W_2(y) & \rightarrow W_1(y)
\end{align*}
\]

This variant or part of the \( H_2 \)-situation might also be called a multiple reversed updates situation. Two transactions \( T_1 \) and \( T_2 \) have common result-parts, \( x \) and \( y \), each of whose corresponding updates are ordered differently.

From a cyclic graph in Fig. 9.33 we have that

\( H_{10} \neq_c \) Any Permutation of \( T_1 \& T_2 \),

and thus we must deduce that:

\( H_{10} \notin \text{CSR} \)

![Fig. 9.33. CSG(H_{10}).](image)

But from Fig. 9.34 we see that:

\( H_{10} =^\text{nra} T_1 \otimes T_2 =^\text{nra} T_2 \otimes T_1 \)

And from Figs. 9.34a and 9.34b we see that:

\( H_{10} =^\text{xc} T_1 \otimes T_2 \)

\( H_{10} =^\text{yc} T_2 \otimes T_1 \)

Then we may conclude that:

\( H_{10} \in A_{\text{NR SR}} \)
So a multiple reversed updates situation is allowed in the not-read-before-write case.

For $H_{10}$ the difference between the corresponding CSG and the corresponding $A_{HR}SG$ is (per definition) only $WW$-type arcs. In this variant we delete all arcs in a cycle going from CSG to $A_{HR}SG$. Again this corresponds to the subcase from Fig. 9.14 in Section 9.2.1.
SITUATION D'

Now, let us exchange the updates $W_1(x)$ and $W_2(y)$ with retrievals $R_1(x)$ and $R_2(y)$ in the above situation D. We end up with the following two patterns occurring together in a schedule $H_{10}^\prime$:

$$R_1(x) - W_2(x)$$

&

$$R_2(y) - W_1(y)$$

We see that the $H_{10}^\prime$-situation with $x$ different from $y$ mirrors the $H_0$-situation [see B] - only excluding the retrievals $R_2(x)$ and $R_1(y)$. Thus this is once more the double dependent updates situation.

From a cyclic graph in Fig. 9.35 we have that

$H_{10}^\prime \notin_c$ Any Permutation of $T_1 \& T_2$, and thus we must deduce that:

$H_{10}^\prime \notin CSR$

![Fig. 9.35. CSG(H_{10}).](image)

Here from a cyclic graph in Fig. 9.36 we further see that:

* $H_{10}^\prime \notin_{nra}$ Any Permutation of $T_1 \& T_2$

![Fig. 9.36. A_{nra}SG[H_{10}].](image)
And from Figs. 9.36a and 9.36b we see that:

\[ H_{10}^* = x_c T_1 \bowtie T_2 \]
\[ H_{10}^* = y_c T_2 \bowtie T_1 \]

So (from the starred fact) we must now even conclude that:

\[ H_{10}^* \notin A_{NRSR} \]

Thus a double dependent updates situation is not allowed in the not-read-before-write case. This is contrary to in the read-before-write case (see situation B).

And the same applies of course also to the more general recursive dependent updates situation or even the single lost update situation. (The \( H_{10}^* \)-situation with \( x \) equal to \( y \) mirrors the \( H_8^* \)-situation [see B']). We may call all these non-allowed situations - double or recursive dependent updates and single lost update - **multiple connected update(s) situations**.
9.3.3 Examples and Counter-Examples for Read-before-write

Here we continue by analyzing four different schedules for the read-before-write case. Two of these schedules (A and A') will directly correspond to the first two situations A and A' in Section 9.3.1, while the other two (B* and B**) will correspond to extensions of the last two situations B and B' in Section 9.3.1. Anyway, the figures used in this section will directly illustrate the differences between any CSG-graph and the corresponding A_RSG-graph.

Schedule A

First, let us look at the following schedule.

\[
\begin{align*}
H_{11} &= R_1(x) \quad R_1(y)[W_1(y)] \\
T_1: & \quad R_2(x)W_2(x)R_2(y)W_2(y) \\
T_2: & \\
\end{align*}
\]

We see that schedule \( H_{11} \) corresponds to the previous \( H_7 \)-situation from Section 9.3.1.

From a cyclic total graph in Fig. 9.37 we have that

\( H_{11} \neq_c \) Any Permutation of \( T_1 \) & \( T_2 \),

and thus we must deduce that:

\( H_{11} \notin CSR \)

But from the acyclic subgraph in Fig. 9.37 we also see that:

\( H_{11} \equiv_{r, a} T_2 \otimes T_1 \)

(In figures like the one above, a dashed arc represents one or more constraint(s) that will vanish going from a CSG-graph - i.e. a total graph - and to the corresponding A_RSG- or A_NRSG-graph - i.e. the subgraph. And an arc with a bracketed label reflects one or more bracketed action(s) in the corresponding schedule - see also Section 9.3.1).
And from Figs. 9.37a and 9.37b we see that:

\[ H_{11} = x_c \ T_1 \circ T_2 \]
\[ H_{11} = y_c \ T_2 \circ T_1 \]

Then we may conclude that:

\[ H_{11} \in A_R \cdot S_R \]
SCHEDULE $A^*$

Now, let us also look at the following schedule.

$$
\begin{align*}
H_{11}^* &= T_1: R_1(x)W_1(x) \quad T_2: R_1(y)[W_1(y)] \\
         & \quad R_2(x)[W_2(x)]R_2(y)W_2(y)
\end{align*}
$$

We see that schedule $H_{11}^*$ corresponds to the previous $H_7^*$-situation from Section 9.3.1.

From a cyclic total graph in Fig. 9.38 we have that

$H_{11}^* \neq \pi_c$ Any Permutation of $T_1$ & $T_2$,

and thus we must deduce that:

$H_{11}^* \notin \text{CSR}$

Here from the cyclic subgraph in Fig. 9.38 we also further see that:

$^*H_{11}^* \neq \pi_{ra}$ Any Permutation of $T_1$ & $T_2$

And from Figs. 9.38a and 9.38b we see that:

$H_{11}^* =_{xc} T_1 \circ T_2$

$H_{11}^* =_{yc} T_2 \circ T_1$

So (from the starred fact) we must now even conclude that:

$H_{11}^* \notin A_{RSR}$
Fig. 9.38a. \( \text{CSG}_1(H_{11}) \).

Fig. 9.38b. \( \text{CSG}_2(H_{11}) \).
SCHEDULE B

Second, let us look at the following schedule.

\[ H_{12} = \]

\[ T_0: [R_0(z)W_0(z)] \]

\[ T_1: [R_1(z)] R_1(x) R_1(y) W_1(y) \]

\[ T_2: [R_2(z)] R_2(y) R_2(x) W_2(x) \]

We see that schedule \( H_{12} \) corresponds to the previous \( H_8 \)-situation from Section 9.3.1; though with a transaction \( T_0 \) and access to an item \( z \) added.

From a cyclic total graph in Fig. 9.39 we have that

\( H_{12} \not\approx_c \) Any Permutation of \( [T_0,] T_1 \) \& \( T_2 \),

and thus we must deduce that:

\( H_{12} \not\in \) CSR

But from the acyclic subgraph in Fig. 9.39 we also see that:

\[ H_{12} =_{ra} [T_0 \otimes T_1 \otimes T_2] =_{ra} [T_0 \otimes T_2 \otimes T_1] \]

And from Figs. 9.39a, 9.39b and 9.39c we see that:

\[ H_{12}(x) =_{c} T_1(x) \otimes T_2(x) \]

\[ H_{12}(y) =_{c} T_2(y) \otimes T_1(y) \]

\[ H_{12} =_{zc} [T_0 \otimes T_1 \otimes T_2] =_{zc} [T_0 \otimes T_2 \otimes T_1] \]

Then we may conclude that:

\( H_{12} \in A_8\text{SR} \)
Fig. 9.39a. $\text{CSG}(H_{i,p})$.

Fig. 9.39b. $\text{CSG}(H_{i,p})$.

Fig. 9.39c. $\text{CSG}(H_{i,p})$. 
SCHEDULE B^* 

Now, let us also look at the following schedule.

\[ H_{12'} = \\
T_0: [R_0(z)W_0(z)] \\
T_1: [R_1(z)] R_1(x) W_1(x) \\
T_2: [R_2(z)] R_2(x) W_2(x) \]

We see that schedule \( H_{12'} \) corresponds to the previous \( H_8' \)-situation from Section 9.3.1; though with a transaction \( T_0 \), access to an item \( z \) and a \( W_1(x) \)-\( W_2(x) \) ordering added.

From a cyclic total graph in Fig. 9.40 we have that

\( H_{12'} \neq \text{CSR} \)

and thus we must deduce that:

\( H_{12'} \in CSG \)

Here from the acyclic subgraph in Fig. 9.40 we also see that:

\[ H_{12'} =_{r_\alpha} [T_0 \circ] T_1 \circ T_2 =_{r_\alpha} [T_0 \circ] T_2 \circ T_1 \]

Fig. 9.40. \( A_{\alpha}(H_{12}) \) vs. CSG(H_{12}).

And from a cyclic graph in Fig. 9.40a and an acyclic graph in 9.40b we further see that:

\* \( H_{12'}[x] \neq \text{CSR} \) Any Permutation of \( T_1[x] \) & \( T_2[x] \)

\[ H_{12'} =_{z\text{c}} [T_0 \circ] T_1 \circ T_2 =_{z\text{c}} [T_0 \circ] T_2 \circ T_1 \]

So (from the starred fact) we must now even conclude that:

\( H_{12'} \notin A_{\alpha}\text{SR} \)
Fig. 9.40a. $\text{CSG}_{4}(H_{12})$.

Fig. 9.40b. $\text{CSG}_{4}(H_{12})$. 
9.3.4 Examples and Counter-Examples for Not-read-before-write

Now we analyze four different schedules for the not-read-before-write case. Again, two of these schedules (C and C') will directly correspond to the first two situations C and C' in Section 9.3.2, while the other two (D* and D'*') will correspond to extensions of the last two situations D and D' in Section 9.3.2. Once more, the figures used in this section will directly illustrate the differences between any CSG-graph and the corresponding $A_{n^*}$SG-graph.

Schedule C

First, let us look at the following schedule.

$$H_{13} = H_{13} = \begin{array}{l}
T_1: R_1(x)[[W_1(x)]] \\
T_2: W_2(x)W_2(z) \\
T_3: W_3(z)W_3(y)
\end{array}$$

We see that schedule $H_{13}$ corresponds to the previous $H_9$-situation from Section 9.3.2.

From a cyclic total graph in Fig. 9.41 we have that $H_{13} \#_c$ Any Permutation of $T_1$, $T_2$ & $T_3$,

and thus we must deduce that:

$H_{13} \notin$ CSR

But from the acyclic subgraph in Fig. 9.41 we also see that:

$$H_{13} =_{nra} T_3 \pi T_1 \pi T_2$$

![Fig. 9.41. $A_{n^*}SG(H_{13})$ vs. CSG($H_{13}$).](image)

And from Figs. 9.41a, 9.41b and 9.41c we see that:

$$H_{13}[x] =_c T_1[x] \pi T_2[x]$$

$$H_{13}[y] =_c T_3[y] \pi T_1[y]$$

$$H_{13}[z] =_c T_2[z] \pi T_3[z]$$
Then we may conclude that:

\[ H_{13} \in A_{NSR} \]
SCHEDULE C′

Now, let us also look at the following schedule.

\[ H_{13′} = \]
\[ T_1: R_1(x) \quad R_1(y)[W_1(y)] \]
\[ T_2: \quad W_2(x)W_2(y) \]

We see that schedule \( H_{13′} \) corresponds to the previous \( H_9′ \)-situation from Section 9.3.2. Thus schedule \( H_{13′} \) equals schedule \( H_{11} \) from Section 9.3.3 - only excluding the retrievals \( R_2(x) \) and \( R_2(y) \).

From a cyclic total graph in Fig. 9.42 we have that

\[ H_{13′} \notin CSR \]

and thus we must deduce that:

\[ H_{13′} \notin C\]

Here from the cyclic subgraph in Fig. 9.42 we also further see that:

* \( H_{13′} \notin nra \) Any Permutation of \( T_1 \) & \( T_2 \)

![Cyclic Subgraph Diagram](image)

Fig. 9.42. \( A_{nr} = SG(H_{13}) \) vs. CSG(H_{13}).

And from Figs. 9.42a and 9.42b we see that:

\[ H_{13′} =_{xc} T_1 \otimes T_2 \]
\[ H_{13′} =_{yc} T_2 \otimes T_1 \]

So (from the starred fact) we must now even conclude that:

\[ H_{13′} \notin A_{HRSR} \]
Fig. 9.42a. CSG_{4}(H_{13}').

Fig. 9.42b. CSG_{5}(H_{13}).
SCHEDULE D*

Second, let us look at the following schedule.

\[ H_{14} = T_0: [R_0(x)R_0(y)] \]
\[ T_1: W_1(x) \quad W_1(y) \]
\[ T_2: W_2(y) \quad W_2(x) \]

We see that schedule \( H_{14} \) corresponds to the previous \( H_{10} \)-situation from Section 9.3.2; though with a transaction \( T_0 \) added.

From a cyclic total graph in Fig. 9.43 we have that

\[ H_{14} \neq_c \text{ Any Permutation of } [T_0, T_1 \& T_2, \]

and thus we must deduce that:

\[ H_{14} \notin \text{ CSR} \]

But from the acyclic subgraph in Fig. 9.43 we also see that:

\[ H_{14} =_{nra} [T_0 \circ] T_1 \circ T_2 =_{nra} [T_0 \circ] T_2 \circ T_1 \]

![Fig. 9.43. A_{nra}SG(H_{14}) vs. CSG(H_{14}).](image)

(Observe in the above figure the multiple RW\(_y\)-arcs - or the multiple RW\(_x\)-arcs - from one reading transaction to several writing transactions. This was explained through the discussion of the A\(_{NR}\)SG-graph concept in Section 9.2.1).

And from Figs. 9.43a and 9.43b we see that:

\[ H_{14} =_{xc} [T_0 \circ] T_1 \circ T_2 \]
\[ H_{14} =_{yc} [T_0 \circ] T_2 \circ T_1 \]

Then we may conclude that:

\[ H_{14} \in A_{NR}SR \]
Fig. 9.43a. $CSG_{\{H_v\}}$.

Fig. 9.43b. $CSG_{\{H_v\}}$.
SCHEDULE D°*

Now, let us also look at the following schedule.

\[ H_{14'} = \]

\[ T_0: \left[ R_0(x) R_0(y) \right] \]
\[ T_1: \quad R_1(x) \quad W_1(y) \]
\[ T_2: \quad R_2(y) \quad W_2(x) \]

We see that schedule \( H_{14'} \) corresponds to the previous \( H_{10'} \)-situation from Section 9.3.2; though with a transaction \( T_0 \) added. Thus schedule \( H_{14'} \) equals schedule \( H_{12} \) from Section 9.3.3 - only excluding the retrievals \( R_1(y) \) and \( R_2(x) \) and deleting (or changing) access to the item \( z \) (into access to items \( x \) and \( y \)).

From a cyclic total graph in Fig. 9.44 we have that

\[ H_{14'} \not\in \text{CSR} \]

and thus we must deduce that:

\[ H_{14'} \not\in \text{CSR} \]

Here from the cyclic [sub]graph in Fig. 9.44 we also further see that:

* \( H_{14'} \not\in \text{nra} \) Any Permutation of \([T_0,] \quad T_1 \quad T_2\)

\[ \begin{align*}
\text{Fig. 9.44. } & A_{nRSG}(H_{14'}) \text{ vs. } CSG(H_{14'}) \\
T_0 & \quad T_1 \\
\quad y_{kw} & \quad x_{kw} \\
T_1 & \quad T_2 \\
\quad x_{kw} & \quad y_{kw} \\
\end{align*} \]

And from Figs. 9.44a and 9.44b we see that:

\[ H_{14'} = x_c \left[ T_0 \sigma \right] T_1 \sigma T_2 = x_c \sigma T_1 \sigma \left[ T_0 \sigma \right] T_2 \]

\[ H_{14'} = y_c \left[ T_0 \sigma \right] T_2 \sigma T_1 = y_c \sigma T_2 \sigma \left[ T_0 \sigma \right] T_1 \]

So (from the starred fact) we must now even conclude that:

\[ H_{14'} \not\in A_{R\sigma R \sigma} \]
Fig. 9.44a. $\text{CSG}_0(H_{10})$.

Fig. 9.44b. $\text{CSG}_0(H_{14})$. 
9.4 Description of Effects

In this section we will carry out a further and closer investigation and analysis of our new notions and concepts.

9.4.1 Class Facts and Relations

We shall state some facts about our new classes and relate them to each other and to classes covered earlier.

SUMMARY

The discussions in Sections 9.3.1 and 9.3.2 can be summarized in Table 9.6.

<table>
<thead>
<tr>
<th>System Type</th>
<th>Read-before-write</th>
<th>Not-read-before-write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Case</td>
<td></td>
<td></td>
</tr>
<tr>
<td>II)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a) Double/Recursive Inconsistent Retrievals</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>b) Single Inconsistent Retrieval</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>g) Indirect Inconsistent Retrieval(s)</td>
<td>[N/Y]</td>
<td>Y</td>
</tr>
<tr>
<td>I)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a) Single Lost Update</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>b) Double/Recursive Dependent Updates</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>g) Multiple Reversed Updates</td>
<td>[N/Y]</td>
<td>Y</td>
</tr>
</tbody>
</table>

N means: No, the problem case is not allowed to occur
Y means: Yes, the problem case is allowed to occur

Compare the detailed effects shown here with the overall effects given in Table 9.4 from Section 9.2.1. Observe that even though the not-read-before-write column in Table 9.4 has only N-entries - suggesting that no problem is allowed to occur, both the read-before-write and not-read-before-write columns in Table 9.6 do have some Y-entries - indicating that some problems are allowed to occur. Thus non-serializable situations may happen for both the read-before-write and not-read-before-write cases. We just had to go to such a depth as in
Sections 9.3.2 and 9.3.4 to illustrate it for the not-read-before-write case.

The bracketed entries in Table 9.6 correspond to read-before-write situations not explicitly analyzed in Sections 9.3.1 and 9.3.3. The triple-entry for situation IIc follows directly from the definitions and indirectly from the read-before-write coupling. A \( Y_e \)-entry states that the corresponding situation is allowed when we exclude the extra updates mentioned in the treatment of that specific situation. (For situation IIc the extra updates are \( W_1(y) \) and \( W_1(x) \), see situation C in Section 9.3.2). Further the \( Y_1 \)-entry part states that the corresponding situation even including the extra updates will be allowed according to the definition of the associated correctness criteria (i.e. \( C_{WR} \), see Eq. 9.41 in Section 9.2.1). However the \( N_1 \)-entry part states that the corresponding situation including the extra updates even will not be allowed when we also take into account that the supposed application area is restricted to read-before-write (i.e. ending up with \( C_{WR} \cap C_R \)). Likewise, a double-entry for situation Ic indirectly follows from the read-before-write coupling. The \( Y \)-entry part states that the corresponding situation will be allowed according to the definition of the associated correctness criteria (i.e. \( C_{WR} \)). However the \( N \)-entry part states that the corresponding situation will not be allowed when we also take into account that the supposed application area is restricted to read-before-write (i.e. \( C_{WR} \cap C_R \)).

(Obviously, as mentioned in Section 9.1.2, the basic inconsistent retrievals problem is not the only possible problem per multiple item).

The situations discussed in Sections 9.3.1 and 9.3.2 may also be checked against the classes \( C_0^\prime \) - \( C_3^\prime \) defined in Eqs. 9.33a-d in Section 9.1.2. The results are given in Table 9.7. They will be needed in the next two paragraphs.

As indicated in Section 5.3, the \( C_0^\prime \) column is based on the ReLock alternative of Table 5.6 in the same section.

Note that in Table 9.7 (and in Table 9.6) we anticipate that all transactions of a schedule being a member in class \( C_i^\prime \) execute at the same "level" i. See Section 9.1.2. Further observe that the content of Table 9.7 is the same for both the not-read-before-write and read-before-write cases, while the content of Table 9.6 is of course split on these two cases.

From Eqs. 9.44 and 9.48 in Section 9.2.1 we have that:

\[
A_{SR} = C_{WR} \cap C_R
\]

\[
A_{NR-SR} = C_{WR-RW} \cap C_{NR}
\]

Further from Eqs. 9.41 and 9.45 in the same section we see that:

\[
C_{WR} \supset C_{WR-RW}
\]

And from Eqs. 9.32a-b in Section 9.1.2 we see that:

\[
C_R \subset C_{NR}
\]
Table 9.7. Detailed Effects of mentioned Scheduling Criteria.

<table>
<thead>
<tr>
<th>Problem Case</th>
<th>$C_2^{-}$</th>
<th>$C_1^{-}$</th>
<th>$C_3^{-}$</th>
<th>$C_4^{-}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a Double/Recursive Inconsistent Retrievals</td>
<td>Y</td>
<td>[N_1]</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>b Single Inconsistent Retrieval</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>c Indirect Inconsistent Retrieval(s)</td>
<td>Y</td>
<td>[N_1]</td>
<td>[N_1]</td>
<td>N</td>
</tr>
</tbody>
</table>

N means: No, the problem case is not allowed to occur
Y means: Yes, the problem case is allowed to occur

A direct consequence of these statements is that:

$$- [A_{RSR} \& A_{NRSR}] \land [A_{RSR} \not\& A_{NRSR}]$$  \hspace{1cm} (Eq. 9.70)

Thus neither $A_{RSR}$ nor $A_{NRSR}$ is a subclass of the other.

(This partially explains the existence of N-entry parts for situations IIc (inclusive variant) and Ic in the read-before-write column in Table 9.6 even though the corresponding entries in the not-read-before-write column are Y-s).

The mentioned relationships are illustrated in Fig. 9.44'.

The classes $A_{RSR}$ and $A_{NRSR}$ are both basically concurrency control oriented. An orthogonal recovery orientation may be superimposed by an addition of RC-, ACA- or ST-like requirements. (See Sections 2.2.8 and 2.2.9). However in the next two paragraphs our more concurrency control oriented classes will be compared as they are, with both some more concurrency control oriented and some more recovery oriented classes treated earlier.

Two other notions which we ought to compare our new classes with, are the pairwise serializability mentioned in [Schl78] and the weak consistency presented in [Garc82].

The pairwise serializability criterion effectively corresponds to allowing no cycles of length two but allowing all cycles of length
three or higher in a CSG-graph. So while this criterion depends on the number of arcs in a cycle, both our criteria depend on the type of arcs in cycle. With reference to Table 9.6 we have already indicated that a IIa-situation or a Ib-situation may be generalized from two to any number of arcs. Actually all the discussed situations but IIc and Ia have versions with two to any number of arcs. A IIc-situation will have three or more arcs, while a Ia-situation always has just two arcs. We have mainly employed examples with as few arcs as possible out of simplicity reasons. Taking into account both the basic and the generalized versions, it is easy to show that the class of schedules corresponding to pairwise serializability is neither a subclass of, nor equal to nor a superclass of neither A_{SR}R nor A_{NR}SR.

The weak consistency concept will be treated in depth in Section 9.4.2. Effectively it will allow a cycle of length four (or higher) corresponding to a generalized version of a IIb-situation. Two of these arcs reflect write-read conflicts, while two of them reflect read-write conflicts. The two transactions which thus experience inconsistent views, have to be read-only transactions. So while this criterion only allows inconsistent views (i.e. between multiple views) for read-only transactions, one of our criteria even allows inconsistent split views (i.e. within single views) for both read-only and read-write transactions.

Hence it is easy to see a connection between the pairwise serializability of [Schl78] and the weak consistency of [Garc82]. Actually [Schl78] also employs the alternative term "consistency in a weak sense".
READ-BEFORE-WRITE CASE

First, we may relate our new class \( A_R \) SR to different classes of
serializable schedules treated earlier. Note that we do this even
without a notion of a commit in our class. The interesting classes are
class FSR from the Appendix and classes VSR\(_0\), VSR\(_N\) and CSR from
Section 2.2.6. The equations to be given are all stated in a read-
before-write context.

With reference to Eq. 4.51 in Section 4.3 we have that:

\[ \text{FSR} \supset \text{VSR}_0 = \text{VSR}_N = \text{CSR} \]

From Eq. 9.31 in Section 9.1.2 and Eq. 9.44 in Section 9.2.1 we get
that:

\[ A_R \) SR \supset \text{CSR} \]

The discussions in Section 9.3.1 show that even:

\[ A_R \) SR \supset \text{CSR} \]

Hence we may deduce that:

\[ A_R \) SR \supset \text{VSR}_0 = \text{VSR}_N = \text{CSR} \quad \text{(Eq. 9.71)} \]

Referring to Table 9.6 and applying the FSG-construct from the
Appendix, we may conclude that schedules corresponding to a 1B-
situation (inclusive variant) or a 1b-situation will both be members
of \( A_R \) SR without being members of FSR. This proves that:

\[ A_R \) SR \notin \text{FSR} \]

We have to refer to the \( H_2 \)-schedule from the Appendix. Applying the
FSG-construct from the Appendix and the \( A_R \) SG-construct from Section
9.2.1, we may conclude that the \( H_2 \)-schedule will be a member of FSR
without being a member of \( A_R \) SR. (Note that it is necessary to
distinguish between live and dead actions and not only between live
and dead transactions to assure membership in FSR. Retrieval \( R_2(x) \)
is a dead action, while all the other retrievals and updates are live
actions). This proves that:

\[ A_R \) SR \notin \text{FSR} \]

Hence we may deduce that:

\[ [A_R \) SR \notin \text{FSR}_T] \land [A_R \) SR \notin \text{FSR}_T] \quad \text{(Eq. 9.72a)} \]

The above equation concerns the set of all valid schedules. Thus
theoretically neither \( A_R \) SR nor FSR is a subclass of the other. This
stems from the possibility of dead retrieval actions occurring.

However, as long as a wander-transaction continues to read items in a
skeleton-database, it will have to write at least one more item. This
stems from the semantics of wander-transactions accessing a skeleton-
database. (Once more consult Section 9.5). Thus there will practically be no dead retrieval actions. The below equation reflects this, as FSR now cannot contain a member that $A_{R}SR$ does not contain.

\[ A_{R}SR \supset FSR_{P} \quad \text{(Eq. 9.72b)} \]

From Section 4.3 we even get that:

\[ FSR_{P} = VSR_{0} \]

Hence it follows that:

\[ FSR_{P} = VSR_{0} = VSR_{N} = CSR \]

The shown relationships are illustrated in Fig. 9.44^a.

![Diagram illustrating relationships between classes (Fig. 9.44^a)](image)

But now recall the multiversion serializable classes corresponding to CSR and VSR$_{0}$. These classes, MCSR and MVSR$_{0}$, were introduced in Section 9.1.2. From Eq. 9.33e in the same section and the fact that MVSR$_{0} = MCSR$ for the read-before-write case, it is easy to show that:

\[
[A_{R}SR \not\in MCSR] \land [A_{R}SR \not\in MCSR] \\
[A_{R}SR \not\in MVSR_{0}] \land [A_{R}SR \not\in MVSR_{0}] 
\]

Second, we may relate our new class $A_{R}SR$ to different classes of non-serializable schedules treated earlier. The interesting classes are classes $C_{0} - C_{3}$ in Section 9.1.2, classes $C_{0} - C_{3}$ in Section 5.3 and classes $C_{A} - C_{C}$ in Section 2.3.2.
Initially, let us look at the associated correctness criteria (i.e. \(C_{WR}\), again see Eq. 9.41 in Section 9.2.1) without taking into account that the supposed application area is restricted to read-before-write.

From Eqs. 9.33a-d in Section 9.1.2 and Table 5.3 in Section 5.3 we get that:

\[
C_0' \geq C_1' \\
C_1' \geq C_2' \geq C_3'
\]

Hence it follows that:

\[
C_0' \geq C_1' \geq C_2' \geq C_3'
\]

The content of Table 9.7 shows that even:

\[
C_0' \geq C_1' \geq C_2' \geq C_3'
\]

Adding Eq. 9.41 in Section 9.2.1 we get that:

\[
C_{WR} \leq C_0' \\
C_{WR} \geq C_3'
\]

The contents of Tables 9.7 and 9.6 show that even:

\[
C_{WR} \leq C_0' \\
C_{WR} \geq C_3'
\]

Hence we may deduce that:

\[
- C_0' \geq C_{WR} \geq C_3' \quad \text{(Eq. 9.73)}
\]

Referring to Tables 9.6 and 9.7, we may conclude that schedules corresponding to an IIC-situation (inclusive variant) or a I c-situation will both be members of \(C_{WR}\) without being members of \(C_1'\). Remember that the Y-entry parts apply in Table 9.6 when we leave out the supposed application area. This proves that:

\[
C_{WR} \not\geq C_1'
\]

Again referring to Tables 9.7 and 9.6, we may conclude that a schedule corresponding to a Ia-situation will be a member of \(C_2'\) without being a member of \(C_{WR}\). This proves that:

\[
C_{WR} \not\geq C_2'
\]

Hence we may deduce that:

\[
- [C_{WR} \not\geq C_1'] \land [C_{WR} \not\geq C_1'] \quad \text{(Eq. 9.74a)}
\]

\[
- [C_{WR} \not\geq C_2'] \land [C_{WR} \not\geq C_2'] \quad \text{(Eq. 9.74b)}
\]
Thus neither \( C_{WR} \) nor \( C_1 \) is a subclass of the other. And neither \( C_{WR} \) nor \( C_2 \) is a subclass of the other.

From Table 5.3 in Section 5.3 and Eqs. 9.33a-d in Section 9.1.2 we also get that:

\[
C_0 = C_0^\prime \\
C_3 \subseteq C_3^\prime
\]

Further from the same table and equations we get that:

\[
C_1 \subseteq C_1^\prime \\
C_2 \subseteq C_2^\prime
\]

And from Table 5.6 (the ReLock alternative) in Section 5.3 we know that the Ia-situation which will be allowed by \( C_2^\prime \) without being allowed by \( C_{WR} \), still is allowed by \( C_2 \) (and \( C_1 \)).

Hence we may deduce that the relations between \( C_{WR} \) and \( C_0 \) - \( C_3 \) respectively will be the same as between \( C_{WR} \) and \( C_0^\prime \) - \( C_3^\prime \). See Eqs. 9.73 and 9.74a-b.

From Eqs. 5.1 and 5.2 in Section 5.3 we have that \( C_B = C_2 \) and \( C_C = C_3 \). The relations between \( C_{WR} \) and \( C_B - C_C \) immediately follow from these equalities.

Further it is easy to show that:

\[
C_A \supseteq C_{WR}
\]

To have equations valid for all the three alternatives in Section 5.3 - i.e. ReLock, Upgrade Lock and One-Lock, we may at least deduce that:

\[
C_0 \not\subseteq C_{WR} \\
C_A \not\subseteq C_{WR} \\
C_1 \not\subseteq C_{WR} \\
C_2 [=C_B] \not\subseteq C_{WR} \\
C_3 [=C_C] \subseteq C_{WR}
\]

These equations are even valid when we go from LUE-variants to 2PL-variants. (See Sections 2.3.2 and 5.3).

Finally, let us look at the associated correctness criteria also taking into account that the supposed application area is restricted to read-before-write (i.e. ending up with \( C_{WR} \cap C_R \)).

With reference to Eq. 9.44 in Section 9.2.1 we have effectively that:

\[
A_{SR} = C_{WR} \cap C_R
\]

Hence it follows that:

\[
A_{SR} \subseteq C_{WR}
\]
We have to recall Eq. 9.67 in Section 9.2.2. It says that for a write-write pattern

$$- W_i(w) - W_j(w) \quad \text{(Eq. 9.75)}$$

occurring in a schedule, there must also be a corresponding read-write pattern occurring in the schedule in the read-before-write case:

$$\Rightarrow R_i(w) - W_j(w)$$

For the chosen local criterion (i.e. CMX, see Eq. 9.43 in Section 9.2.1) there must also even be a corresponding write-read pattern occurring in the schedule in the read before-write case:

$$\Rightarrow W_i(w) - R_j(w)$$

Thus there can be no legal total cycle corresponding to WW-relationships only (refer to $C_1'$) or to WR-relationships only (refer to $C_2'$) without there also being a global cycle corresponding to only WR-relationships (refer to $A_{R SR}$). Legal means observing the local criterion and observing the read-before-write restriction.

Hence we may conclude that:

$$C_{G:WR} \cap C_{L:WR-RW-WW} \cap C_R \subseteq C_{G:WR-WW} \subseteq C_{G:WW}$$

This means that

$$C_{WR} \cap C_R \subseteq C_2' \subseteq C_1',$$

which again means that:

$$A_{R SR} \subseteq C_2'$$

Referring to Tables 9.7 and 9.6, we may conclude that a schedule corresponding to a Ia-situation will still be a member of $C_2'$ without being a member of $A_{R SR}$. This shows that even:

$$A_{R SR} \subseteq C_2'$$

Remember now that the N-entry parts apply for a IIc-situation (inclusive variant) and a Ic-situation in Table 9.6 when we do not leave out the supposed application area. These are further the only changes in the first column from superimposing the read-before-write restriction. This shows that still:

$$A_{R SR} \supset C_3' \cap C_R$$

Hence we may deduce that:

$$- C_0' \supset C_1' \supset C_2' \supset A_{R SR} \supset C_3' \quad \text{(Eq. 9.76)}$$

An $\cap C_R$ should be added to $C_3'$ in this equation. However the equation is valid both without and with an $\cap C_R$ on the other primed classes.
The deduced relationships are illustrated in Fig. 9.44″b.

The relations between $A_{RSR}$ and $C_0$, $C_A$ or $C_3 [=C_2]$ respectively will be the same as between $C_{WR}$ and $C_0$, $C_A$ or $C_3 [=C_0]$. The relations between $A_{RSR}$ and $C_1$ or $C_2 [=C_2]$ are more difficult to prove. But the above five equations simultaneously covering all the three alternatives in Section 5.3 - i.e. ReLock, Upgrade Lock and One-Lock, are still valid for both the LUE- and 2PL-variants. Thus at least we have the corresponding weak relations between $A_{RS}$ and $C_1$ or $C_2 [=C_0]$.

Note that the difference between Eqs. 9.73 + 9.74a-b (which describe the chosen global and local criteria) and Eq. 9.76 reflects the added read-before-write restriction of the associated application area. Further observe that as well a deletion of the read-before-write restriction as a deletion of the local criterion would bring us from relations of the Eq. 9.76-type to relations of the Eqs. 9.73 + 9.74a-b -type. See also Fig. 9.44″″b in the next paragraph.
NOT-READ-BEFORE-WRITE CASE

First, we may also relate our new class $A_{NR}$ to different classes of serializable schedules treated earlier. The equations to be given are all stated in a not-read-before-write context.

With reference to Eq. 4.50 in Section 4.3 we have that:

$$\text{FSR} \supset \text{VSR}_0 \supset \text{VSR}_N \supset \text{CSR}$$

From Eq. 9.31 in Section 9.1.2 and Eq. 9.48 in Section 9.2.1 we get that:

$$A_{NR} \supset \text{CSR}$$

The discussions in Section 9.3.2 show that even:

$$- A_{NR} \supset \text{CSR} \quad \text{(Eq. 9.77)}$$

Referring to Table 9.6 and applying the FSG-construct from the Appendix, we may conclude that schedules corresponding to a IIC-situation (inclusive variant) or a Ic-situation will both be members of $A_{NR}$ without being members of FSR. This proves that:

$$A_{NR} \nsubseteq \text{FSR}$$

We have to recall the $H_3$'-schedule from Section 2.2.9. From figures like Fig. 4.8 in Section 4.2.3 and Fig. 4.15 in Section 4.2.5, we may conclude that the $H_3$'-schedule will be a member of $\text{VSR}_N$ without being a member of $A_{NR}$ (even modulo commit). This proves that:

$$A_{NR} \nsubseteq \text{VSR}_N$$

Hence we may deduce that:

$$- [A_{NR} \nsubseteq \text{FSR}] \land [A_{NR} \nsubseteq \text{FSR}] \quad \text{(Eq. 9.78a)}$$

$$- [A_{NR} \nsubseteq \text{VSR}_0] \land [A_{NR} \nsubseteq \text{VSR}_0] \quad \text{(Eq. 9.78b)}$$

$$- [A_{NR} \nsubseteq \text{VSR}_N] \land [A_{NR} \nsubseteq \text{VSR}_N] \quad \text{(Eq. 9.78c)}$$

Thus neither $A_{NR}$ nor FSR is a subclass of the other. Further neither $A_{NR}$ nor $\text{VSR}_0$ is a subclass of the other. And neither $A_{NR}$ nor $\text{VSR}_N$ is a subclass of the other.

The shown relationships are illustrated in Fig. 9.44''a.

But again recall the multiversion serializable classes corresponding to CSR and $\text{VSR}_0$. These classes, $\text{MCSR}$ and $\text{MVS}_0$, were introduced in Section 9.1.2. From Eq. 9.33e in the same section and the fact that
M\textsubscript{VSR\textsubscript{0}} \supset M\textsubscript{CSR} for the not-read-before-write case, it is easy to show that:

\[ A_{N_{R_{SR}}} \subseteq M\textsubscript{CSR} \subseteq M\textsubscript{VSR\textsubscript{0}} \]

Second, we may also relate our new class \( A_{N_{R_{SR}}} \) to different classes of non-serializable schedules treated earlier.

Initially, let us look at the associated correctness criteria (i.e. \( C_{WR-RW} \), see Eq. 9.45 in Section 9.2.1).

Recall the relations that were deduced in the previous paragraph independent of any specific correctness criteria.

Adding Eq. 9.45 in Section 9.2.1 we get that:

\[ C_{WR-RW} \leq C_{0} \]
\[ C_{WR-RW} \geq C_{3} \]

The contents of Tables 9.7 and 9.6 show that even:

\[ C_{WR-RW} \subseteq C_{0} \]
\[ C_{WR-RW} \supseteq C_{3} \]

Hence we may deduce that:

\[ C_{0} \supseteq C_{WR-RW} \supseteq C_{3} \]  \hspace{1cm} (Eq. 9.79)

Referring to Tables 9.6 and 9.7, we may conclude that schedules corresponding to a IIc-situation (inclusive variant) or a Ic-situation
will both be members of $C_{WR-RW}$ without being members of $C_1^-$. This proves that:

$$C_{WR-RW} \not\subseteq C_1^-$$

Again referring to Tables 9.7 and 9.6, we may conclude that schedules corresponding to a IIb-situation, a Ia-situation or a Ib-situation will all be members of $C_2^-$ without being members of $C_{WR-RW}$. This proves that:

$$C_{WR-RW} \not\supseteq C_2^-$$

Hence we may deduce that:

- $[C_{WR-RW} \not\subseteq C_1^-] \land [C_{WR-RW} \not\subseteq C_1^-]$ (Eq. 9.80a)
- $[C_{WR-RW} \not\subseteq C_2^-] \land [C_{WR-RW} \not\subseteq C_2^-]$ (Eq. 9.80b)

Thus neither $C_{WR-RW}$ nor $C_1^-$ is a subclass of the other. And neither $C_{WR-RW}$ nor $C_2^-$ is a subclass of the other.

As in the previous paragraph, the relations between $C_{WR-RW}$ and $C_0$, $C_1$, $C_2 [\equiv C_B]$ or $C_3 [\equiv C_C]$ respectively will be the same as between $C_{WR-RW}$ and $C_0$, $C_1^-$, $C_2^-$ or $C_3^-$. See Eqs. 9.79 and 9.80a-b.

Further it is still easy to show that:

$$C_A \supseteq C_{WR-RW}$$

To have equations valid for all the three alternatives in Section 5.3 - i.e. ReLock, Upgrade Lock and One-Lock, we may still at least deduce that:

$$C_0 \not\subseteq C_{WR-RW}$$
$$C_A \not\subseteq C_{WR-RW}$$
$$C_1 \not\subseteq C_{WR-RW}$$
$$C_2 [\equiv C_B] \not\subseteq C_{WR-RW}$$
$$C_3 [\equiv C_C] \subseteq C_{WR-RW}$$

These equations are even valid when we go from LUE-variants to 2PL-variants. (See Sections 2.3.2 and 5.3).

Finally, let us look at the associated correctness criteria taking into account that any possible application area is supposed to be not-read-before-write (i.e. ending up with $C_{WR-RW} \cap C_{NR}$).

With reference to Eq. 9.48 in Section 9.2.1 we have effectively that:

$$A_{MR-SR} = C_{WR-RW} \cap C_{NR}$$
Hence it follows that:

\[ A_{NR}SR = C_{WR-RW} \]

Thus all the relations stated or indicated for \( C_{WR-RW} \) in this paragraph also apply for \( A_{NR}SR \).

We have specifically that:

- \( C_0' \supset A_{NR}SR \supset C_3' \)  
  \[ \text{(Eq. 9.81)} \]

- \[ [A_{NR}SR \not\subset C_1'] \land [A_{NR}SR \not\subset C_1'] \]  
  \[ \text{(Eq. 9.82a)} \]

- \[ [A_{NR}SR \not\subset C_2'] \land [A_{NR}SR \not\subset C_2'] \]  
  \[ \text{(Eq. 9.82b)} \]

The deduced relationships are illustrated in Fig. 9.44''b.
9.4.2 Further Exploration of Read-Before-Write Case

Here, let us look at the class defined for read-before-write and try to characterize its corresponding properties.

**Basis**

From Eq. 9.31 in Section 9.1.2 and Eqs. 9.43 and 9.44 in Section 9.2.1 we may again state the obvious fact that:

\[ CMX \supset A_k \supset CSR \]

Thus in any schedule \( H \) where an item \( x \) is retrieved and updated by some transactions, any of these transactions \( T_k \) will necessarily have the following view:

\[ R_i(x)W_i(x) - R_j(x)W_j(x) - R_k(x)W_k(x) - R_l(x)W_l(x) - R_m(x)W_m(x) \]

This corresponds to a serial execution per single item - like in the operating systems model in Section 2.1.2. From \( T_k \)'s point of view its retrieval of \( x \) sees the effect from the updates on that item of some transactions \( (T_i, T_j) \) and sees no effect from the updates on that item of the other transactions \( (T_l, T_m) \). The perception of the transaction updates sequence with regard to item \( x \) will be the same for all transaction retrievals, even for those belonging to transactions not updating item \( x \) later. However this perceived (and real) sequence may vary for different items. Such effects correspond to class CMX.

But in any schedule \( H^- \) where items are retrieved and updated by some transactions, any of these transactions \( T_k \) will not necessarily have the following view:

\[ R_iW_i - R_jW_j - R_kW_k - R_lW_l - R_mW_m \]

(The \( R_kW_k \)-notation has no connection to the special model in Chapter 4. It is only a shorthand notation for a complete transaction).

This corresponds to a serializable execution per multiple item - like in the database systems model in Section 2.1.2. From \( T_k \)'s point of view its retrievals of items would see the effects from the updates on all those items of some transactions \( (T_i, T_j) \) and would see no effects from the updates on any of those items of the other transactions \( (T_l, T_m) \). The perception of the transaction updates sequence with regard to each item should be the same for all transaction retrievals, even for those belonging to transactions not updating any item at all. So this perceived (though virtual) sequence should even be the same for different items. (See Fig. 2.4 from Section 2.2.2). Such effects correspond to class CSR.
With reference to the above descriptions we want to examine more closely which situations and schedules class $A_b$SR actually allows to occur between the two extremes illustrated by Eqs. 9.84 and 9.83.
REVERSED VIEWS

First, let us start from a schedule $H_{15}$ in which the following two patterns occur together:

\[ R_3(x)W_3(x) - R_1(x)W_1(x) - R_2(x)W_2(x) \]

\&

\[ R_2(y)W_2(y) - R_1(y)W_1(y) - R_3(y)W_3(y) \]

From transaction $T_1$'s point of view its retrieval of $x$ sees the effect on that item from the update of transaction $T_3$, but not from the update of transaction $T_2$. Further from transaction $T_1$'s point of view its retrieval of $y$ sees the effect on that item from the update of transaction $T_2$, but not from the update of transaction $T_3$. So the update-sequences with regard to items $x$ and $y$ (as perceived by the transaction) are different, or the views (of the transaction) with regard to items $x$ and $y$ are reversed.

Is this $H_{15}$-situation allowed? Or rather; what parts of this situation will be allowed to occur?

We will base our discussion on the assumption that all retrievals are to remain.

It is easy to see that there are no single item problems (corresponding to situation $B'$ in Section 9.3.1).

From situations $A$ and $B$ in Section 9.3.1 we know that any latest retrieval of any item may always be accompanied by an update on the same item. Thus updates $W_2(x)$ and $W_3(y)$ may always remain.

But from situation $A'$ in Section 9.3.1 we may deduce that updates $W_1(x)$ and $W_2(y)$ may not stay together. The same applies to the pair of updates $W_3(x)$ and $W_1(y)$, and even to the pair of updates $W_3(x)$ and $W_2(y)$.

Hence if $W_3(x)$ stays, then $W_2(y)$ and $W_1(y)$ have to be omitted. The part of $H_{15}$ corresponding to this will be denoted $H_{151}$. (Alternatively if $W_2(y)$ stays, then $W_3(x)$ and $W_1(x)$ have to be omitted).

Finally if neither $W_3(x)$ nor $W_2(y)$ stays, then no other actions have to be omitted. The part of $H_{15}$ corresponding to this will be denoted $H_{152}$.

Thus we end up with two basic variant-cases:
$H_{151}$:

\[ R_3(x)W_3(x) - R_1(x)W_1(x) - R_2(x)W_2(x) \]

&

\[ R_2(y) - R_1(y) - R_3(y)W_3(y) \]

(Alternatively: Include $W_2(y)$ & $W_1(y)$ and Exclude $W_3(x)$ & $W_1(x)$)

$H_{152}$:

\[ R_3(x) - R_1(x)W_1(x) - R_2(x)W_2(x) \]

&

\[ R_2(y) - R_1(y)W_1(y) - R_3(y)W_3(y) \]
SITUATION $E_1$

Let us now illustrate the $H_{151}$-situation through our usual graphical test procedure.

From a cyclic graph in Fig. 9.45 we have that

$H_{151} \not\in c$ Any Permutation of $T_1$, $T_2$ & $T_3$,

and thus we must deduce that:

$H_{151} \notin CSR$

![Fig. 9.45. CSG($H_{151}$).]

But from Fig. 9.46 we see that:

$H_{151} \equiv \gamma_\alpha \circ T_3 \circ T_1 \circ T_2$

![Fig. 9.46. A$_5$SG($H_{151}$).]
And from Figs. 9.46a and 9.46b we see that:

\[ H_{151} = x_c T_3 \circ T_1 \circ T_2 \]

\[ H_{151} = y_c T_1 \circ T_2 \circ T_3 = y_c T_2 \circ T_1 \circ T_3 \]

Then we may conclude that:

\[ H_{151} \in A_{R,SR} \]
SCHEDULE $E_1$

A schedule corresponding to the $H_{151}$-situation is the following $H_{15}^{-}$:

$$H_{15}^{-} =$$

$T_1$: \[ R_1(y)R_1(x)W_1(x) \]

$T_2$: \[ R_2(y) \]

$T_3$: \[ R_2(x)W_2(x) \]

\[ R_3(x)W_3(x) \]

\[ R_3(y)W_3(y) \]
SITUATION $E_2$

Let us also illustrate the $H_{152}$-situation through our usual graphical test procedure.

From a cyclic graph in Fig. 9.47 we have that

$H_{152} \neq_c$ Any Permutation of $T_1$, $T_2$ & $T_3$,

and thus we must deduce that:

$H_{152} \notin CSR$

![Diagram Fig. 9.47. CSG($H_{152}$).]

But from Fig. 9.48 we see that:

$H_{152} =_r a T_1 \odot T_2 \odot T_3 =_r a T_1 \odot T_3 \odot T_2$

![Diagram Fig. 9.48. $A_{CS}(H_{152})$.]
And from Figs. 9.48a and 9.48b we see that:

\[ H_{152} =_{xc} T_3 \circ T_1 \circ T_2 \]

\[ H_{152} =_{yc} T_2 \circ T_1 \circ T_3 \]

Then we may conclude that:

\[ H_{152} \in A_{RSR} \]
SCHEDULE $E_2$

A schedule corresponding to the $H_{152}$-situation is the following $H_{15}^{**}$:

$$H_{15}^{**} =
\begin{align*}
T_1: & \quad R_1(y)W_1(y)R_1(x)W_1(x) \\
T_2: & \quad R_2(y)R_2(x)W_2(x) \\
T_3: & \quad R_3(x)R_3(y)W_3(y)
\end{align*}$$
REVERSED VIEWS REVISITED

Second, let us continue with a schedule \( H_{16} \) in which the following two/four patterns occur together:

\[
\begin{align*}
R_4(y)W_4(y) - R_1(y)W_1(y) & \quad \& \quad R_1(x)W_1(x) - R_3(x)W_3(x) \\
& \quad \& \quad R_3(x)W_3(x) - R_2(x)W_2(x) & \quad \& \quad R_2(y)W_2(y) - R_4(y)W_4(y)
\end{align*}
\]

From transaction \( T_1 \)'s point of view its retrieval of \( y \) sees the effect on that item from the update(s) of transaction(s) \( T_4 \) (and \( T_2 \)), and its retrieval of \( x \) does not see the effect on that item from the update(s) of transaction(s) \( T_3 \) (and \( T_2 \)). Further from transaction \( T_2 \)'s point of view its retrieval of \( x \) sees the effect on that item from the update(s) of transaction(s) \( T_3 \) (and \( T_1 \)), and its retrieval of \( y \) does not see the effect on that item from the update(s) of transaction(s) \( T_4 \) (and \( T_1 \)). So the update-sequences (with regard to the set of items) as perceived by transactions \( T_1 \) and \( T_2 \) are different, or the views of transactions \( T_1 \) and \( T_2 \) (with regard to the set of items) are reversed.

Again: is this \( H_{16} \)-situation allowed? Or rather; what parts of this situation will be allowed to occur?

We will base our discussion on the assumption that all retrievals still are to remain and that updates \( W_4(y) \) and \( W_3(x) \) also are to remain.

Once more it is easy to see that there are no single item problems (corresponding to situation B' in Section 9.3.1).

From situations A and B in Section 9.3.1 we still know that any latest retrieval of any item may always be accompanied by an update on the same item. Thus updates \( W_1(y) \) and \( W_2(x) \) may always remain.

And from situation A' in Section 9.3.1 we may again conclude about updates \( W_1(x) \) and \( W_2(y) \). If \( T_4 \) and \( T_3 \) are not the same transaction, then either \( W_1(x) \) or \( W_2(y) \) - but not both - may stay. Further if \( T_4 \) and \( T_3 \) are the same transaction, then neither \( W_1(x) \) nor \( W_2(y) \) can stay.

Hence if \( W_1(x) \) stays, then \( W_2(y) \) has to be omitted - and \( T_4 \) has to be different from \( T_3 \). The part of \( H_{16} \) corresponding to this will be denoted \( H_{161} \). (Alternatively if \( W_2(y) \) stays, then \( W_1(x) \) has to be omitted - and \( T_4 \) still has to be different from \( T_3 \)).

Finally if neither \( W_1(x) \) nor \( W_2(y) \) stays, then \( T_4 \) may be equal to \( T_3 \). The part of \( H_{16} \) corresponding to this will be denoted \( H_{162} \).

Thus we end up with two basic subcases:
$H_{161}$:

\[ R_4(y)W_4(y) - R_1(y)W_1(y) \quad \& \quad R_1(x)W_1(x) - R_3(x)W_3(x) \]

\[ \& \]

\[ R_3(x)W_3(x) - R_2(x)W_2(x) \quad \& \quad R_2(y) \quad - \quad R_4(y)W_4(y) \]

(Alternatively: Include $W_2(y)$ and Exclude $W_1(x)$)

$H_{162}$:

\[ R_3(y)W_3(y) - R_1(y)W_1(y) \quad \& \quad R_1(x) \quad - \quad R_3(x)W_3(x) \]

\[ \& \]

\[ R_3(x)W_3(x) - R_2(x)W_2(x) \quad \& \quad R_2(y) \quad - \quad R_3(y)W_3(y) \]

We see that the $H_{162}$-situation mirrors the $H_{152}$-situation - only with $T_3$ and $T_1$ interchanged.
SITUATION F₁

Let us now illustrate the $H_{₁₆₁}$-situation through our usual graphical test procedure.

From a cyclic graph in Fig. 9.49 we have that

$H_{₁₆₁} \neq \_c$ Any Permutation of $T₁, T₂, T₃$ & $T₄$,

and thus we must deduce that:

$H_{₁₆₁} \notin CSR$

![Fig. 9.49. CSG($H_{₁₆₁}$).](image)

But from Fig. 9.50 we see that:

$H_{₁₆₁} =_{r,a} T₄ \circ T₁ \circ T₃ \circ T₂$

![Fig. 9.50. AₙSG($H_{₁₆₁}$).](image)
And from Figs. 9.50a and 9.50b we see that:

\[
\begin{align*}
    H_{161}(x) &= c \cdot T_1(x) \circ T_3(x) \circ T_2(x) \\
    H_{161}(y) &= c \cdot T_2(y) \circ T_4(y) \circ T_1(y)
\end{align*}
\]

![Diagram](image)

**Fig. 9.50a. CSG_4(H_{161}).**

![Diagram](image)

**Fig. 9.50b. CSG_4(H_{161}).**

Then we may conclude that:

\[ H_{161} \in A_{R SR} \]
SCHEDULE $F_1$

A schedule corresponding to the $H_{161}$-situation is the following $H_{16}^-$:

$$H_{16}^- =
\begin{align*}
T_1: & \quad R_1(x)W_1(x) & R_1(y)W_1(y) \\
T_2: & \quad R_2(x)R_2(y) & W_2(x) \\
T_3: & \quad R_3(x)W_3(x) \\
T_4: & \quad R_4(y)W_4(y)
\end{align*}$$
SITUATION $F_2$

Let us also illustrate the $H_{162}$ situation through our usual graphical test procedure.

From a cyclic graph in Fig. 9.51 we have that

$H_{162} \neq c$ Any Permutation of $T_1$, $T_2$ & $T_3$,

and thus we must deduce that:

$H_{162} \notin CSR$

![Fig. 9.51. CSG($H_{162}$).]

But from Fig. 9.52 we see that:

$H_{162} =_{ra} T_3 \circ T_1 \circ T_2 =_{ra} T_3 \circ T_2 \circ T_1$

![Fig. 9.52. A$_a$SG($H_{162}$).]
And from Figs. 9.52a and 9.52b we see that:

\[ H_{162} =_{x_c} T_1 \circ T_3 \circ T_2 \]

\[ H_{162} =_{y_c} T_2 \circ T_3 \circ T_1 \]

Then we may conclude that:

\[ H_{162} \in A_{R,S,R} \]
SCHEDULE $F_2$

A schedule corresponding to the $H_{162}$-situation is the following $H_{16''}$:

$$H_{16''} =$$

$T_1$: $R_1(x)$

$T_2$: $R_2(x)R_2(y)$

$T_3$: $R_3(x)W_3(x)$

$R_1(y)W_1(y)$

$W_2(x)$

$R_3(y)W_3(y)$

Naturally we see that schedule $H_{16''}$ "equals" schedule $H_{15''}$ - only with $T_3$ and $T_1$ interchanged.
GENERALIZATION

To deepen our understanding of the qualities of class AₘSR, we need to clarify the inherent properties of the situations shown to be allowed in this section.

First, let us illustrate the actual views of the interesting transactions in the two basic subcases allowed, which are derived from the non-allowed H₁₆-situation.

\( H₁₆₁: \)

<table>
<thead>
<tr>
<th>T₁'s View:</th>
<th>T₄ \rightarrow T₁[?y!] T₁[?x!] \rightarrow T₃ \rightarrow T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₂'s View:</td>
<td>T₁ \rightarrow T₃ \rightarrow T₂[?x!] T₂[?y] \rightarrow T₄ \rightarrow T₁</td>
</tr>
</tbody>
</table>

\( H₁₆₂: \)

<table>
<thead>
<tr>
<th>T₁'s View:</th>
<th>T₃[?y] \rightarrow T₁[?y!] T₁[?x] \rightarrow T₃[?x] \rightarrow T₂[?x]</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₂'s View:</td>
<td>T₃[?x] \rightarrow T₂[?x!] T₂[?y] \rightarrow T₃[?y] \rightarrow T₁[?y]</td>
</tr>
</tbody>
</table>

(The T[?x]- and T[?x!]-notations are different variants of the T[?x]-notation from Section 9.1.1 reflecting an item projection of a transaction. Both represent the part of transaction T that accesses item x. While T[?x] indicates retrieval only, T[?x!] indicates both retrieval and update.

We have left out any concatenation symbols as we have allowed the separate parts of a transaction to change order in some cases. This is done to be able to associate each part with (the parts of) the other transactions making up the transaction's view.

Second, let us illustrate the actual views of the interesting transactions in the two basic variant-cases allowed, which are derived from the non-allowed H₁₅-situation.

\( H₁₅₂: \)

<table>
<thead>
<tr>
<th>T₃'s View:</th>
<th>T₁[?y] \rightarrow T₃[?y!] T₃[?x] \rightarrow T₃[?x] \rightarrow T₂[?x]</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₂'s View:</td>
<td>T₁[?x] \rightarrow T₂[?x!] T₂[?y] \rightarrow T₁[?y] \rightarrow T₃[?y]</td>
</tr>
</tbody>
</table>
$H_{151}$:

T₃'s View: $T_3[?y!] T_3[?x!] - T_1[x] - T_2[x]$

T₂'s View: $T_3[x] - T_1[x] - T_2[?x!] T_2[?y] - T_3[y]$

Initially, the $H_{161}$-illustration shows that two (or generally more) different transactions - $T_1$ and $T_2$ - are allowed to have reversed (or generally inconsistent) perceptions of an execution order of all (or generally some) of the other transactions - $T_3$ and $T_4$.

If all the transactions that may experience reversed (or inconsistent) views were read-only transactions, then this case would comply with the weak-consistency concept presented in [Garc82]. Weak-consistency means that any schedule of only read-write transactions (i.e. all eventual read-only transactions have been omitted) has to be serializable. Thus each eventual read-only transaction will see a set of consistent values when retrieving any set of items. But different read-only transactions may observe different execution orders as the basis for their different sets of consistent values. Or paraphrased; all read-write transactions plus any read-only transaction will necessarily correspond to a serializable schedule, but two or more read-write transactions plus two or more read-only transactions will not necessarily correspond to a serializable schedule.

But $A_pSR$ does not restrict the transactions that may experience inconsistent views to be read-only. We see that both transactions $T_1$ and $T_2$ (in addition to transactions $T_3$ and $T_4$ of course) are read-write in the $H_{161}$-situation.

The $H_{161}$-situation also illustrates the fact that in a schedule any specific transaction is always allowed to update an item as long as all other transactions updating the same item precede the specific transaction according to that ones view. This emerges in the $H_{161}$-illustration as $T_1[?y!]$ and $T_2[?x!]$.

The $H_{161}$-situation even illustrates the fact that among the transactions experiencing inconsistent views, some may sometimes - but not always - also update an item even though one is not the last to update this item among the currently executing transactions. This emerges in the $H_{161}$-illustration as $T_1[?x!]$ - but only $T_2[?y]$.

The view in the $H_{161}$-situation of either transaction $T₁$ or transaction $T₂$ (those that experience inconsistent views) may in general be formulated as the one of transaction $T_k$ in schedule $H$ below:

- $H$: (Eq. 9.85)

$$R_1(w)W_1(w), R_j(w)W_j(w) - B_k(r), B_k(w)W_k(w) - R_1(r)W_1(r), R_m(r)W_m(r)$$

$T_k$ is thus the last transaction to retrieve and update item $w$ (and the first transaction to retrieve item $r$). From $T_k$'s point of view its retrieval of $w$ sees the effect from the updates on that item of
transactions $T_i$ and $T_j$, and its retrieval of $r$ sees no effect from the updates on that item of transactions $T_i$ and $T_m$. The perception of the transaction updates sequences (i.e. whether a specific transaction sees another transaction as preceding or succeeding itself) with regard to items $r$ and $w$ may always be inconsistent for at least the separate $T_k$s corresponding to the different possible interpretations of $r$ and $w$. However the perception of the internal sequence of $T_i$ and $T_j$ or of $T_i$ and $T_m$ (i.e. of transactions possibly updating several common items) must be consistent for different $T_k$s irrespective of whether a specific $T_k$ sees $T_i$ and $T_j$ or $T_i$ and $T_m$ as preceding or succeeding itself. This stems from the read-before-write coupling of $A_{R,SR}$.

Even adding $W_k(r)$ sometimes - but not always - results in a $T_k$ that still experiences an execution order inconsistent with those of the transactions of the original type - without the corresponding $H$ being non-allowed by $A_{R,SR}$.

Alternatively, deleting $W_k(w)$ - corresponding to converting to a read-only transaction - always results in a $T_k$ that still has a view inconsistent with those of the transactions of the original type. Such inconsistent viewing will be allowed by $A_{R,SR}$ for any read-only $T$ - i.e. not necessarily only those that retrieve a $w$ last (and an $r$ first). This special case corresponds exactly to the weak-consistency notion covered above.

Compare Eq. 9.85 and this current discussion with Eqs. 9.84 and 9.83 and their corresponding comments.

Further, the $H_{162}$-illustration shows that a specific transaction - $T_1$ - is even allowed to have a split perception with regard to whether another transaction - $T_3$ - precedes or succeeds itself. Hence part of a transaction - $T_3[y]$ - may seem to precede oneself, and part of it - $T_3[x]$ - may seem to succeed oneself.

The $H_{162}$-illustration also shows that two (or generally more) different transactions - $T_1$ and $T_2$ - are further allowed to have reversed (or generally inconsistent) perceptions with regard to whether specific parts - $T_3[x]$ and $T_3[y]$ - of a (or generally several) transaction(s) - $T_3$ - separately either precedes or succeeds itself.

The $H_{162}$-situation even illustrates the refined fact that in a schedule any specific transaction is always allowed to update an item as long as all other transaction parts updating the same item precede the specific transaction according to that ones view. This emerges in the $H_{162}$-illustration as $T_1[?y!]$ and $T_2[?x!]$.

Finally, the $H_{152}$-illustration again supports our earlier statement that the $H_{152}$-situation mirrors the $H_{162}$-situation - only with $T_3$ and $T_1$ interchanged. And the $H_{151}$-illustration also supports our earlier indication that the $H_{151}$-situation is just a variant of the $H_{152}$-situation.

The refined view in the $H_{162}$-situation (alternatively the $H_{152}$- / $H_{151}$-situation) of either transaction $T_1 (T_3)$ or transaction $T_2 (T_2)$ (those that experience a split view - and inconsistent split views)
may in general be formulated as the one of transaction $T_k$ in schedule $H'$ below:

$$H': \quad (Eq. 9.86)$$

$$R_i(w)\ W_i(w)\ R_m(w)\ W_m(w) - R_k(r)\ R_k(w)\ W_k(w) - R_i(r)\ W_i(r)\ R_m(r)\ W_m(r)$$

$T_k$ thus contains the last transaction part to retrieve and update item $w$ (and the first transaction part to retrieve item $r$). From $T_k$'s point of view its retrieval of $w$ sees the effect from the updates on that item of parts of some transactions $T_i$ and $T_m$, and its retrieval of $r$ sees no effect from the updates on that item of parts of the same transactions $T_i$ and $T_m$. The perception of the transaction part updates sequences (i.e. whether a specific transaction sees another transaction's part as preceding or succeeding itself) with regard to items $r$ and $w$ may always be inconsistent for at least the separate $T_k$s corresponding to the different possible interpretations of $r$ and $w$. However the perception of the internal sequence of some $T_i$- and $T_m$-parts (i.e. of transaction parts possibly updating several common items) must be consistent for different $T_k$s irrespective of whether a specific $T_k$ sees the given $T_i$- and $T_m$-parts as preceding or succeeding itself. This stems from the read-before-write coupling of $A_{SR}$.

Again deleting $W_k(w)$ - corresponding to converting to a read-only transaction - always results in a $T_k$ that still has a split view - and a split view inconsistent with those of the transactions of the original type. Such inconsistent split viewing will be allowed by $A_{SR}$ for any read-only T - i.e. not necessarily only those with parts that retrieve a $w$ last (and an $r$ first). This special case corresponds actually to a generalized version of the weak-consistency notion covered above.

Also compare Eq. 9.86 and this current discussion with Eqs. 9.84 and 9.83 and their corresponding comments.

We will now relate the discussed notions.

The CSR class corresponds to always maintaining the database consistent. It assures any view of a set of retrieved items for any read-write or read-only transaction both to be internally consistent (i.e. within single views) and externally consistent (i.e. between multiple views). Thus any schedule of transactions is serializable.

The weak-consistency concept of [Garc82] also corresponds to always maintaining the database consistent. It assures any view of a set of retrieved items for any read-write transaction both to be internally consistent (i.e. within single views) and externally consistent (i.e. between multiple views). But it only assures any view of a set of retrieved items for any read-only transaction to be internally consistent (i.e. within single views). So omitting all read-only transactions from any schedule of transactions makes it serializable.

The $A_{SR}$ class of ours does not correspond to maintaining the database consistent. (That any integrity constraints lack, makes consistency preservation irrelevant as correctness criterion). It does not assure a view of a set of retrieved items for any read-write or read-only transaction either to be internally consistent or externally
consistent. (That some overall semantics information exists, makes alternative requirements relevant as correctness criterion). However omitting all read-only actions (i.e. initial retrievals of specific items that are not accompanied by final updates on the same items) from any schedule of transactions makes it serializable. This stems once more from the read-before-write coupling of ARSR.

We will also illustrate the discussed notions.

First, let us repeat the $H_{161}$-situation.

$$H_{161}:$$

\[
R_4(y)W_4(y) - R_1(y)W_1(y) \quad \& \quad R_1(x)W_1(x) - R_3(x)W_3(x)
\]

\[
\&
\]

\[
R_3(x)W_3(x) - R_2(x)W_2(x) \quad \& \quad R_2(y) - R_4(y)W_4(y)
\]

The corresponding schedule that we showed was:

$$H_{161}' =$$

\[
T_1: \ R_1(x)W_1(x)
\]

\[
T_2: \ R_2(x)W_2(y)
\]

\[
T_3: \ R_3(x)W_3(x)
\]

\[
T_4: \ R_4(y)W_4(y)
\]

From our analysis of the $H_{161}$-situation (Figs. 9.49 - 9.50b) we also have that:

$$H_{161}' \in ARSR \quad [H_{161}' =_{RA} T_4 \circ T_3 \circ T_2]$$

but

$$H_{161}' \notin CSR$$

Now omitting all read-only actions - i.e. $R_2(y)$ - from $H_{161}'$, we end up with the following schedule:

$$H_{161}'_{RW} =$$

\[
T_1: \ R_1(x)W_1(x)
\]

\[
T_2: \ R_2(x)
\]

\[
T_3: \ R_3(x)W_3(x)
\]

\[
T_4: \ R_4(y)W_4(y)
\]

For this schedule it is straightforward to deduce that:

$$H_{161}'_{RW} \in CSR \quad [H_{161}'_{RW} =_{C} T_4 \circ T_3 \circ T_2]$$
Then instead, let us delete updates $W_1(x), W_1(y)$ and $W_2(x)$ from the original $H_{161}$-situation. We end up with the following situation.

$$H_{161}:$$
$$R_4(y)W_4(y) - R_1(y) \quad & \quad R_1(x) - R_3(x)W_3(x)$$

$$\&$$

$$R_3(x)W_3(x) - R_2(x) \quad & \quad R_2(y) - R_4(y)W_4(y)$$

A corresponding schedule is:

$$H_{161}^{*} =$$
$$T_1: R_1(x) \quad & \quad R_1(y)$$
$$T_2: R_2(x)R_2(y)$$
$$T_3: R_3(x)W_3(x)$$
$$T_4: R_4(y)W_4(y)$$

This is a typical example of a weak-consistent schedule as described above.

Now further omitting all read-only transactions - i.e. $T_1$ and $T_2$ - from $H_{161}^{*}$, we end up with the following schedule:

$$H_{161}^{* *RW} =$$
$$T_3: R_3(x)W_3(x)$$
$$T_4: R_4(y)W_4(y)$$

For this schedule it is once more straightforward to deduce that:

$$H_{161}^{* *RW} \in CSR \quad [H_{161}^{* *RW} = c_{T_3}OT_4]$$

Second, let us repeat the $H_{162}$-situation.

$$H_{162}:$$
$$R_3(y)W_3(y) - R_1(y)W_1(y) \quad & \quad R_1(x) - R_3(x)W_3(x)$$

$$\&$$

$$R_3(x)W_3(x) - R_2(x)W_2(x) \quad & \quad R_2(y) - R_3(y)W_3(y)$$

The corresponding schedule that we showed was:

$$H_{162}^{*} =$$
$$T_1: R_1(x) \quad & \quad R_1(y)W_1(y)$$
$$T_2: R_2(x)R_2(y) \quad & \quad W_2(x)$$
$$T_3: R_3(x)W_3(x) \quad & \quad R_3(y)W_3(y)$$
From our analysis of the $H_{162}$-situation (Figs. 9.51 - 9.52b) we also have that:

$$H_{16}'' \in A_{SR} \quad [H_{16}'' =_{ra} T_3 \circ T_1 \circ T_2 =_{ra} T_3 \circ T_2 \circ T_1]$$

but

$$H_{16}'' \notin CSR$$

Now omitting all read-only actions - i.e. $R_1(x)$ and $R_2(y)$ - from $H_{16}''$, we end up with the following schedule:

$$H_{16}''_{RW} =$$

$$T_1: \quad R_1(y)$$

$$T_2: \quad R_2(x)$$

$$T_3: \quad R_3(x)W_3(x) \quad R_3(y)W_3(y)$$

For this schedule it is straightforward to deduce that:

$$H_{16}''_{RW} \in CSR \quad [H_{16}''_{RW} =_{c} T_3 \circ T_1 \circ T_2 =_{c} T_3 \circ T_2 \circ T_1]$$

Then instead, let us delete updates $W_1(y)$ and $W_2(x)$ from the original $H_{162}$-situation. We end up with the following situation.

$$H_{164}:$$

$$R_3(y)W_3(y) \rightarrow R_1(y) \quad \& \quad R_1(x) \rightarrow R_3(x)W_3(x)$$

$$\&$$

$$R_3(x)W_3(x) \rightarrow R_2(x) \quad \& \quad R_2(y) \rightarrow R_3(y)W_3(y)$$

A corresponding schedule is:

$$H_{16}'''' =$$

$$T_1: \quad R_1(x) \quad R_1(y)$$

$$T_2: \quad R_2(x)R_2(y)$$

$$T_3: \quad R_3(x)W_3(x) \quad R_3(y)W_3(y)$$

Compared to a weak-consistent schedule as described above - i.e. representing just inconsistent views of read-only (and even read-write) transactions, this is a typical example of what might be called a super-weak-consistent schedule - i.e. representing even inconsistent split views of read-only transactions.
Now further omitting all read-only transactions - i.e. $T_1$ and $T_2$ - from $H_{16}$, we end up with the following schedule:

$$H_{16} \Rightarrow RW = T_3: R_3(x)W_3(x)R_3(y)W_3(y)$$

For this schedule it is once more straightforward to deduce that:

$$H_{16} \Rightarrow RW \in CSR \ [H_{16} \Rightarrow RW = c T_3]$$

To round off, we have to repeat the definitions of classes $A_{SR}$ and CSR from respectively Eq. 9.44 in Section 9.2.1 and Eq. 9.31 in Section 9.1.2:

$$A_{SR} = C_{WR} \cap C_R$$

$$CSR = C_{WR-RW-WW} \cap C_R$$

With reference to Eq. 9.30 (and Table 9.2) in Section 9.1.2, what may we say about the classes between the two extreme ones listed; i.e. all classes observing the local criterion and at least corresponding to the binary relation WR(Any Schedule) being a partial order?

Exploiting the equalities given in Eqs. 9.34 to 9.36 in Section 9.2.1, we can initially group the above-mentioned classes according to their global criteria only. This reflects the consequences of Eq. 9.67 in Section 9.2.2; i.e. the effects from the read-before-write restriction on different global criteria. The results are as follows:

I) $G:WR$

II) $G:WR,WW / G:WR-WW$


IV) $G:WR-WW,RW$


For the remaining four classes from Table 9.2 we also get:


Adding the consequences of Eq. 9.75 in Section 9.4.1, we can finally group the above-mentioned classes according to their combined global and local criteria. Remember that Eq. 9.75 is an extension of Eq. 9.67 which generate effects from the read-before-write restriction and a specific local criterion on combinations of different global criteria and the specific local criterion. The two classes in group II will be equal, and this common class will also be equal to the class in group
I. Further the class in group IV will be equal to the common class in group III. Hence the results are as follows:

I) \( C_{WR} = C_{WR,WW} = C_{WR-WW} \)

III) \( C_{WR,RW} = C_{WR,RW,WW} = C_{RW-WW,WR} = C_{WR-WW,RW} \)

V) \( C_{WR-RW} = C_{WR-RW,WW} = C_{WR-RW-WW} \)

For the remaining four classes from Eq. 9.30 we still get:

\( C_{WW} / C_{RW} = C_{RW,WW} = C_{RW-WW} \)

Note that the equality of classes \( C_{WR} \) and \( C_{WR-WW} \) in group I does not mean that our class \( A_{SR} \) is equal to the class \( C_{2} \) of Section 9.1.2. Remember the distinction between a combined global + local criterion and a pure global criterion. With reference to Eq. 9.76 in Section 9.4.1 we actually have that:

\[ A_{SR} = C_{WR} \cap C_{R} = C_{WR-WW} \cap C_{R} \subseteq C_{G:WR-WW} \cap C_{R} = C_{2} \cap C_{R} \]

Further with reference to Eq. 9.74b in the same section we also have that:

\[ [C_{WR} \not\subseteq C_{G:WR-WW}] \land [C_{WR} \not\subseteq C_{G:WR-WW}] \]

We will cover all three groups I), III) and V).

I) This group may be represented by

\( C_{WR} = A_{SR} \).

From the analysis in Section 9.3.3 (Figs. 9.37 - 9.37b and 9.39 - 9.39c) we know that:

\( H_{11} \in C_{WR} \)

\( H_{12} \in C_{WR} \)

Likewise from this section we know that:

\( H_{16} \in C_{WR} \)

\( H_{16} \in C_{WR} \)

Further it is straightforward to see that:

\( H_{16} \in C_{WR} \)

\( H_{16} \in C_{WR} \)
III) This group may be represented by

\[ C_{WR, RW, WW} \]

Through an analysis like in Section 9.3.3 we may deduce that:

\[ H_{11}(\text{excl. } W_1(y)) \in C_{WR, RW, WW} \]
\[ H_{11}(\text{incl. } W_1(y)) \notin C_{WR, RW, WW} \]
\[ H_{12} \notin C_{WR, RW, WW} \]

Likewise we may deduce that:

\[ H_{16} \notin C_{WR, RW, WW} \]
\[ H_{16}'' \notin C_{WR, RW, WW} \]

Further, deleting

\[ W_1(y) \lor (R_1(x) \land W_1(x)) \lor W_2(x) \lor R_2(y) \]

from \( H_{16}'' \) will lead to membership in the class.

Likewise, deleting

\[ (W_1(y) \lor R_1(x)) \land (W_2(x) \lor R_2(y)) \]

from \( H_{16}''' \) will lead to membership in the class.

Thus, for example for \( H_{16}''' \) (deleting \( W_1(y) \), \( W_2(x) \) and \( W_3(x) \) from \( H_{16}''' \) - i.e. more than necessary) and \( H_{16}'''' \) (deleting \( W_1(y) \) and \( W_2(x) \) from \( H_{16}''' \) - i.e. not more than necessary) we have that:

\[ H_{16}''' \in C_{WR, RW, WW} \]
\[ H_{16}'''' \in C_{WR, RW, WW} \]

V) This group will be represented by

\[ C_{WR-RW-WW} = CSR. \]

From the analysis in Section 9.3.3 we also know that:

\[ H_{11} \notin C_{WR-RW-WW} \]
\[ H_{12} \notin C_{WR-RW-WW} \]

Likewise from this section we know that:

\[ H_{16} \notin C_{WR-RW-WW} \]
\[ H_{16}'' \notin C_{WR-RW-WW} \]
Further it is straightforward to see that:

$$H_{16} \subseteq \mathcal{C}_{WR-RW-WW}$$
$$H_{16} \subseteq \mathcal{C}_{WR-RW-WW}$$

Thus the covered classes (representing the groups) are proper subclasses of each other. Further group III) corresponds to the notion of weak-consistent views of read-only and read-write transactions plus its extension super-weak-consistent views of read-only transactions. Naming this class $\mathcal{C}_{SWC}$ (for super-weak-consistency), we have the following relationships:

- $A_{R}SR \supseteq \mathcal{C}_{SWC} \supseteq CSR$ \hspace{1cm} (Eq. 9.87)

It is also possible to prove some further non-trivial relationships among typical classes representing all the resulting groups. We even include the class CMX and a class $\mathcal{C}_{WC}$ corresponding to the notion of weak-consistent views of read-only transactions; i.e. reflecting the weak-consistency concept of [Garc02].

$$\mathcal{C}_{CMX} \supseteq \mathcal{C}_{WW}$$
$$\mathcal{C}_{WW} \supseteq \mathcal{C}_{WR} [A_{R}SR] \supseteq \mathcal{C}_{WR,RW} [A_{R}SR]$$
$$\mathcal{C}_{WW} \supseteq \mathcal{C}_{RW} \supseteq \mathcal{C}_{WR,RW} [A_{R}SR]$$
$$[\mathcal{C}_{WR} [A_{R}SR] \neq \mathcal{C}_{RW}] \land [\mathcal{C}_{WR} [A_{R}SR] \neq \mathcal{C}_{RW}]$$
$$\mathcal{C}_{SWC} [A_{R}SR] \supseteq \mathcal{C}_{WR,RW} \supseteq \mathcal{C}_{SWC} \supseteq CSR [A_{R}SR]$$

The previously given descriptive comparison of the classes CSR, $\mathcal{C}_{WC}$ and $A_{R}SR$ concentrated on the possibilities; i.e. the freedom inherent in the specifications of the different classes. With reference to Table 9.6 in Section 9.4.1 we will also include a more direct comparison of the classes $A_{R}SR$, $\mathcal{C}_{SWC}$ and CSR. This concentrates on the effects; i.e. the constraints deducible from the specifications of the specific classes.

- $A_{R}SR$ allows no double/recursive inconsistent retrievals and allows no single lost update

- $\mathcal{C}_{SWC}$ observes the same constraints as $A_{R}SR$ plus allows no single inconsistent retrieval - inclusive variant; i.e. allows no double/recursive dependent updates

Remember that a double dependent updates situation is a subcase of the inclusive variant of a single inconsistent retrieval situation, see Section 9.3.1.

- CSR observes the same constraints as $\mathcal{C}_{SWC}$ plus allows no single inconsistent retrieval - exclusive variant (i.e. neither the inclusive nor the exclusive variant)
9.4.3 Further exploration of Not-read-before-write Case

Now, let us look at the class defined for not-read-before-write and try to characterize its corresponding properties.

(As mentioned in Sections 9.2.1 and 9.3.2 our treatment of the not-read-before-write case will not be as complete and general as for the read-before-write case. Thus the situations to be analyzed do not represent an exhaustive exploration of the possibilities, but more a basic illustration of the qualities).

**Basis**

From Eq. 9.31 in Section 9.1.2 and Eqs. 9.47 and 9.48 in Section 9.2.1 we may again state the obvious fact that:

\[ \text{CMX} \triangleright A_{NR SR} \triangleright \text{CSR} \]

Thus in any schedule \( H \) where an item \( x \) is retrieved and updated by some transactions, any of these transactions \( T_k \) will **necessarily** have the following view:

\[ - H: \quad W(x) - R(x) - W(x) - W(x) - W(x) \]

(Eq. 9.88)

This corresponds again to a serial execution per **single item** - like in the operating systems model in Section 2.1.2. Compared to Eq. 9.83 in Section 9.4.2 the only difference is of course that an update must not necessarily be preceded by a retrieval on the same item. These effects correspond once more to class CMX.

But in any schedule \( H' \) where items are retrieved and updated by some transactions, any of these transactions \( T_k \) will **not necessarily** have the following view:

\[ - H': \quad R_i W_i - R_j W_j - R_k W_k - R_l W_l - R_m W_m \]

(Eq. 9.89)

(Again the \( R_k W_k \)-notation has no connection to the special model in Chapter 4. It is only a shorthand notation for a complete transaction).

This corresponds again to a **serializable execution per multiple item** - like in the database systems model in Section 2.1.2. Compared to Eq. 9.84 in Section 9.4.2 there is no difference. These effects correspond once more to class CSR.

With reference to the above descriptions we want to examine more closely which situations and schedules class \( A_{NR SR} \) actually allows to occur between the two extremes illustrated by Eqs. 9.89 and 9.88.
REVERSED UPDATES

Let us use a schedule $H_{17}$ in which the following four patterns occur together:

$$R_1(x) - W_3(x)$$
\&
$$W_3(x) - W_4(x)$$
\&
$$W_4(y) - W_3(y)$$
\&
$$W_3(y) - R_2(y)$$

With regard to item $x$ an update of transaction $T_3$ is followed by the update of transaction $T_4$. Further with regard to item $y$ an update of transaction $T_4$ is followed by the update of transaction $T_3$. So the update-sequences with regard to items $x$ and $y$ are reversed.

This $H_{17}$-situation will be allowed to occur in its entirety.
SITUATION 6

Let us here illustrate the $H_{17}$-situation through our usual graphical test procedure.

From a cyclic graph in Fig. 9.53 we have that

$H_{17} \neq c$ Any Permutation of $T_1$, $T_2$, $T_3$ & $T_4$,

and thus we must deduce that:

$H_{17} \notin$ CSR

![Fig. 9.53. CSG($H_{17}$).]

But from Fig. 9.54 we see that:

$H_{17} =_{nra} T_1 \otimes T_3 \otimes T_4 \otimes T_2 =_{nra} T_1 \otimes T_4 \otimes T_3 \otimes T_2$

![Fig. 9.54. A_nraSG($H_{17}$).]
(As for Fig. 9.43 from Section 9.3.4 observe in the above figure the multiple $RW_x$-arcs from one reading transaction to several writing transactions - and the multiple $WR_y$-arcs from several writing transactions to one reading transaction. This was explained through the discussion of the $A_{WSG}$-graph concept in Section 9.2.1).

And from Figs. 9.54a and 9.54b we see that:

\[
\begin{align*}
H_{17}[x] &= T_1[x] \otimes T_3[x] \otimes T_4[x] \\
H_{17}[y] &= T_4[y] \otimes T_3[y] \otimes T_2[y]
\end{align*}
\]

Fig. 9.54a. $CSG(H_{17})$.

Fig. 9.54b. $CSG(H_{17})$.

Then we may conclude that:

\[ H_{17} \in A_{WSR} \]
SCHEDULE 6

A schedule corresponding to the $H_{17}$-situation is the following $H_{17}^*$:

\[
H_{17}^* = \\
T_1: \quad R_1(x) \\
T_2: \quad R_2(y) \\
T_3: \quad W_3(x), \quad W_3(y) \\
T_4: \quad W_4(x)W_4(y)
\]
GENERALIZATION

To deepen our understanding of the qualities of class $A_{WR}$, we need to clarify the inherent properties of the situations shown to be allowed in this section.

Let us illustrate the actual update-sequences of the items in the basic case allowed, which is covered in the $H_{17}$-situation.

\[H_{17}:\]
\[
x\text{'s Update-sequence: } T_3[x!] \rightarrow T_4[x!]
\]
\[
y\text{'s Update-sequence: } T_4[y!] \rightarrow T_3[y!]
\]

(Again the $T[x!]$-notation is a variant of the $T[x]$-notation from Section 9.1.1 reflecting an item projection of a transaction. It represents the part of transaction $T$ that accesses item $x$, and $T[x!]$ indicates update only).

The $H_{17}$-illustration shows that the values of two (or generally more) different items - $x$ and $y$ - are allowed to correspond to reversed (or generally inconsistent) execution orders of all (or generally some) of the updating transactions - $T_3$ and $T_4$.

The views in the $H_{17}$-situation of both transaction $T_1$ and transaction $T_2$ (those that experience inconsistent update-sequences) may in general be formulated as the one of transaction $T_k$ in schedule $H$ below:

\[H: \quad (Eq. 9.90)\]
\[
R_iW_i \rightarrow R_kW_k \rightarrow R_iW_i
\]
\[
R_jW_j \rightarrow R_mW_m
\]

From $T_k$'s point of view its retrievals of items see the effects from the updates on all those items of transactions $T_1$ and $T_3$ and see no effects from the updates on any of those items of transactions $T_1$ and $T_m$. The perception of the internal sequence of parts of $T_1$ and $T_3$ or of parts of $T_1$ and $T_m$ (i.e. of parts of transactions possibly updating several common items) may be inconsistent for separate $T_k$s or for separate parts of a $T_k$. However the perception of the transaction update sequences (i.e. whether a specific transaction sees another transaction as preceding or succeeding itself) with regard to one or more item(s) must be consistent for different $T_k$s. These two facts are just the opposite of those applying for $A_{FR}$ and given in Eq. 9.85 (and Eq. 9.86) in Section 9.4.2. It stems from the lack of read-before-write coupling and inclusion of both WR-relationships and RW-relationships in $A_{WR}$.

Again compare Eq. 9.90 and this current discussion with Eqs. 9.89 and 9.88 and their corresponding comments.
To round off, we have to repeat the definitions of classes $A_{NR}SR$ and CSR from respectively Eq. 9.48 in Section 9.2.1 and Eq. 9.31 in Section 9.1.2:

$$A_{NR}SR = C_{WR-RW} \cap C_{WR}$$  
$$CSR = C_{WR-RW-WW} ([\cap C_{WR}])$$  

With reference to Eq. 9.30 (and Table 9.2) in Section 9.1.2, what may we say about the classes between the two extreme ones listed; i.e. all classes observing the local criterion and at least corresponding to the binary relation WR-RW(Any Schedule) being a partial order?

There are actually only three groups of one class each for the not-read-before-write case:

I) $C_{WR-RW}$

II) $C_{WR-RW,WW}$

III) $C_{WR-RW-WW}$

Observe that the specifications of the three classes/groups mentioned are the same as the specifications of the three classes constituting group V) for the read-before-write case in Section 9.4.2.

I) This group equals the single class

$$C_{WR-RW} = A_{NR}SR.$$  

From the analysis in Section 9.3.4 (Figs. 9.41 - 9.41c and 9.43 - 9.43b) we know that:

$$H_{13} \in C_{WR-RW}$$  
$$H_{14} \in C_{WR-RW}$$

Likewise from this section we know that:

$$H_{17} \notin C_{WR-RW}$$

II) This group equals the single class

$$C_{WR-RW,WW}.$$  

Through an analysis like in Section 9.3.4 we may deduce that:

$$H_{13}(\text{excl. } W_{1}(y) \lor W_{1}(x)) \in C_{WR-RW,WW}$$  
$$H_{13}(\text{incl. } W_{1}(y) \land W_{1}(x)) \notin C_{WR-RW,WW}$$  
$$H_{14} \notin C_{WR-RW,WW}$$
Likewise we may deduce that:

\[ \text{Let } H_{17}' \notin C_{\text{WR-RW,WW}} \]

III) This group equals the single class

\[ C_{\text{WR-RW-WW}} = \text{CSR.} \]

From the analysis in Section 9.3.4 we also know that:

\[ H_{13} \notin C_{\text{WR-RW-WW}} \]
\[ H_{14} \notin C_{\text{WR-RW-WW}} \]

Likewise from this section we know that:

\[ H_{17}' \notin C_{\text{WR-RW-WW}} \]

Thus the covered classes (representing the groups) are proper subclasses of each other. Naming the class constituting group II) 
\( C_{\text{FRFW}} \) (for freeze-read-freeze-write - shortly to be explained why), we have the following relationships:

\[ \text{A}_{\text{HRSR}} \supset C_{\text{FRFW}} \supset \text{CSR} \quad \text{(Eq. 9.91).} \]

There are no other non-trivial relationships among the classes from Eq. 9.30 (and Table 9.2).

We will now relate the discussed notions.

With reference to Table 9.6 in Section 9.4.1 (and Section 9.3.2) we will first give a direct comparison of the classes \( \text{A}_{\text{HRSR}}, C_{\text{FRFW}} \) and CSR. This concentrates on the effects; i.e. the constraints deducible from the specifications of the specific classes.

- \( \text{A}_{\text{HRSR}} \) allows no direct inconsistent retrieval(s)
  (i.e. no single inc. retrieval or double/recursive inc. retrievals)
  and allows no multiple connected update(s)
  (i.e. no double/recursive dep. updates or single lost update)

- \( C_{\text{FRFW}} \) observes the same constraints as \( \text{A}_{\text{HRSR}} \)
  plus allows no indirect inconsistent retrieval(s) - incl. variant;
  i.e. allows no multiple reversed updates

Remember that a multiple reversed updates situation is a subcase of the inclusive variant of an indirect inconsistent retrieval situation, see Section 9.3.2.

- CSR observes the same constraints as \( C_{\text{FRFW}} \)
  plus allows no indirect inconsistent retrieval(s) - excl. variant
  (i.e. neither the inclusive nor the exclusive variant)
We will then also include a more descriptive comparison of the same classes.

The $A_{NR}$SR class effectively makes all the multiple retrievals of each single transaction function as one unique atomic unit with regard to the updates of all other transactions. Thus this concept corresponds to freezing the database state during the reading of a specific transaction (upon which the decisions are to be based).

The $C_{FRFW}$ class effectively makes – separately – all the multiple retrievals of each single transaction function as one unique atomic unit with regard to the updates of all other transactions – or all the multiple updates of the same transaction function as one (other) unique atomic unit with regard to the updates and retrievals of all other transactions. Thus this concept corresponds to first freezing the database state during the reading of a specific transaction (upon which the decisions are to be based) – and then later freezing the database during the writing of the same transaction (through which the decisions are to be implemented).

The CSR class effectively makes all the multiple retrievals and updates of each single transaction together function as one unique atomic unit with regard to the retrievals and updates of all other transactions. Thus this concept corresponds to freezing the database (state) totally during the reading, decision-making and writing of a specific transaction.

(The above given characterizations of $A_{NR}$SR and $C_{FRFW}$ are not totally complete. The description of $A_{NR}$SR only reflects the basic prohibition against direct inconsistent retrieval(s) situations, while the description of $C_{FRFW}$ also reflects the added prohibition against inclusive variants of indirect inconsistent retrieval(s) situations. But neither description reflects the basic prohibition against multiple connected update(s) situations. Hence the above characterizations of $A_{NR}$SR and $C_{FRFW}$ are not strong enough. As given they would both allow any schedule in the single-action-multi-item model, see Chapter 4).

Neither $A_{NR}$SR nor $C_{FRFW}$ guarantees a consistent database, but both $A_{NR}$SR and $C_{FRFW}$ guarantee consistent views. (In Section 9.4.2 both $A_{SR}$ and $C_{SVC}$ guarantee neither a consistent database nor consistent views). While CSR (like in Section 9.4.2) guarantees both a consistent database and consistent views.
9.5 Resulting System Examples

In this section we will give some examples showing the functioning of wander-transactions in a skeleton-database. As mentioned in Section 9.2.1 this corresponds to our read-before-write case. Thus we will only deal with schedules that are members/non-members of $A_{SR}$.

9.5.1 Generic Examples

Here we start with four different examples illustrating the synchronization applied in the concurrency control of multiple transactions. The schedules in two of the examples ($B^*$ and $A^*$) will be variants of schedules used earlier in Section 9.3.3, while the schedules in the other two examples ($F_2^*$ and $E_1^*$) will be variants of schedules used earlier in Section 9.4.2. With reference to Section 8.1.3 all the schedules to be covered correspond to true parallel behaviour. Thus they all deal with departures from one-after-the-other atomicity. (See also Section 8.2.2).

Example $B^*$

First, let us imagine the skeleton-database containing the following two item-variants.

\begin{verbatim}
\textbf{x:}$
\begin{array}{l}
\text{Price} = 10 \text{ }$ \\
\text{Quality} = 95 \% \\
\text{No-Left} = 1
\end{array}
\end{verbatim}

\begin{verbatim}
\textbf{y:}$
\begin{array}{l}
\text{Price} = 5 \text{ }$ \\
\text{Quality} = 90 \% \\
\text{No-Left} = 1
\end{array}
\end{verbatim}

(With reference to Eq. 9.26 and Fig. 9.7 in Section 9.1.2, different item-variants represent the one and only item existing at different sites. Or paraphrased; each specific triple corresponds to a specific department store's offer of the one and only article. The price- and quality-attributes of an article-offer should be self-explanatory, while the no-left-attribute indicates that any specific department store may have several instances of the article at the given price and quality).
We will now look at the following two wander-transactions.

\[ T_1: \text{Acquire One: Lowest-Price \& Quality } \geq 85\% \]

\[ T_2: \text{Acquire One: Price } \leq 15\text{\$ \& Highest-Quality} \]

(The meaning of each such statement is that the corresponding transaction, after checking - i.e. first reading - all the available articles, should acquire - i.e. then write - the one article fulfilling the given condition [the best way]).

A system schedule corresponding to the indicated wander-transactions accessing the indicated item-variants, is:

\[ H_{18} = T_1: R_1(x) \quad R_1(y) \quad W_1(y) \]
\[ T_2: \quad R_2(x) \quad R_2(y) \quad W_2(x) \]

We see that schedule \( H_{18} \) resembles schedule \( H_{12} \) from Section 9.3.3. One change is the deletion of transaction \( T_0 \) and access to item-variant \( z \). Further retrievals \( R_2(y) \) and \( R_2(x) \) have changed positions to allow both transactions to read all item-variants in the same sequence - i.e. only action-moves without any consequences with regard to a CSG-graph.

(It is not a system requirement that all transactions should read the item-variants in the same sequence. However this will be employed in the examples out of simplicity reasons).

Thus with reference to Table 9.6 in Section 9.4.1 this schedule corresponds to one double dependent updates situation.

From a cyclic total graph in Fig. 9.55 we have that

\[ H_{18} \notin_c \text{ Any Permutation of } T_1 \text{ \& } T_2, \]

and thus we must deduce that:

\[ H_{18} \notin \text{ CSR} \]

But from the acyclic subgraph in Fig. 9.55 we also see that:

\[ H_{18} =_{ra} T_1 \circ T_2 =_{ra} T_2 \circ T_1 \]

And from Figs. 9.55a and 9.55b we see that:

\[ H_{18} =_{xc} T_1 \circ T_2 \]
\[ H_{18} =_{yc} T_2 \circ T_1 \]
Then we may conclude that:

\[ H_{18} \in A_R SR \]

Since \( H_{18} \) is not a member of CSR, this schedule could not of course be created by a 2PhaseLocking mechanism even applying both shared- and exclusive-locks. (Refer to Section 8.2.2).
EXAMPLE A*

Then, let us imagine the skeleton-database containing the following two item-variants.

\[
\begin{array}{l}
\text{x:} \\
\text{Price} = 10 \$ \\
\text{Quality} = 95 \% \\
\text{No-Left} = 1 \\
\hline \\
\text{y:} \\
\text{Price} = 5 \$ \\
\text{Quality} = 90 \% \\
\text{No-Left} = 2 \\
\end{array}
\]

We will now look at the following two wander-transactions.

\[T_1: \]
Acquire One: Lowest-Price & Quality \geq 85 \%

\[T_2: \]
Acquire One: Price \leq 15 \$ & Highest-Quality

A system schedule corresponding to the indicated wander-transactions accessing the given item-variants, is:

\[H_{19} = \]
\[T_1: R_1(x) R_1(y)W_1(y) \]
\[T_2: R_2(x)R_2(y)W_2(x)W_2(y)\]

We see that schedule \(H_{19}\) resembles schedule \(H_1\), including update \(W_1(y)\) from Section 9.3.3. The change is that update \(W_2(x)\) and retrieval \(R_2(y)\) have changed positions to allow both transactions to read all item-variants before making decision(s) - i.e. only an action-move without any consequences with regard to a CSG-graph.

(In the examples employed in this section all the updates of item-variants of each separate wander-transaction will succeed all the retrievals of item-variants of that specific wander-transaction. If no write could ever precede a read in any transaction schedule, the global part of the membership requirements for class \(A_0SR\) would be assured for any system schedule without any synchronization. Hence the
binary relation WR(Any Schedule) would be a partial order per
definition. In the examples employed in Section 9.5.3 we illustrate
that a wander-transaction accessing a skeleton-database often needs to
update an item-variant before it retrieves another item-variant. Note
that this refers to different item-variants, while the read-before-
write restriction refers to a single item-variant. This section
concentrates on the freedom inherent in our correctness criterion,
while Section 9.5.3 concentrates on the constraints inherent in our
correctness criterion).

Thus with reference to Table 9.6 in Section 9.4.1 this schedule
corresponds to one single inconsistent retrieval situation. It is the
"outer" transaction $T_1$ that experiences inconsistent retrievals.

From a cyclic total graph in Fig. 9.56 we have that

$H_{19} \not\in_c \text{Any Permutation of } T_1 \& T_2,$

and thus we must deduce that:

$H_{19} \not\in \text{CSR}$

But from the acyclic subgraph in Fig. 9.56 we also see that:

$H_{19} =_{ra} T_2 \circ T_1$

![Fig. 9.56. A_{SR}(H_{19}) vs. CSR(H_{19}).](image)

And from Figs. 9.56a and 9.56b we see that:

$H_{19} =_{xc} T_1 \circ T_2$

$H_{19} =_{yc} T_2 \circ T_1$

Then we may conclude that:

$H_{19} \in A_{SR}$

If locking (even applying both shared- and exclusive-locks) is to be
used as scheduling mechanism instead of the most general $A_{SR}$-testing,
the specification of transaction $T_1$ has to be changed into for example
the following to make the resulting schedule which is shown, legal.
T₁:
   Acquire One: Price ≤ 7.5 $ & [Highest-Quality]

Still, as H₁₉ is not a member of CSR, the schedule could not of course be created by a 2PhaseLocking mechanism even applying both shared- and exclusive-locks. (Refer to Section 8.2.2).

An exclusion of update W₂(y) from schedule H₁₉ would reflect a deletion of the second part of the specification of transaction T₂. The resulting schedule would now correspond to one double dependent updates situation. Refer to the previous example B².
EXAMPLE $F_2^*$

Then, let us imagine the skeleton-database containing the following two item-variants.

$x$:

<table>
<thead>
<tr>
<th>Price</th>
<th>10 $</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quality</td>
<td>95 %</td>
</tr>
<tr>
<td>No-Left</td>
<td>2</td>
</tr>
</tbody>
</table>

$y$:

<table>
<thead>
<tr>
<th>Price</th>
<th>5 $</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quality</td>
<td>90 %</td>
</tr>
<tr>
<td>No-Left</td>
<td>2</td>
</tr>
</tbody>
</table>

We will now look at the following three wander-transactions.

$T_1$:
Acquire One: Lowest-Price & Quality $\geq 85$ %

$T_2$:
Acquire One: Price $\leq 15$ $\&$ Highest-Quality

$T_3$:
Acquire One: Price $\leq 15$ $\&$ Highest-Quality

A system schedule corresponding to the indicated wander-transactions accessing the given item-variants, is:

$$H_{2,0} = T_1: R_1(x) \quad R_1(y)W_1(y)$$
$$T_2: \quad R_2(x)R_2(y) \quad W_2(x)$$
$$T_3: \quad R_3(x)R_3(y)W_3(x) \quad W_3(y)$$

We see that schedule $H_{2,0}$ resembles schedule $H_{1,0}^*$ ($\approx H_{1,5}^*$ with $T_3$ and $T_1$ interchanged) from Section 9.4.2. The change is that retrieval $R_3(y)$ has been moved in between retrieval $R_3(x)$ and update $W_3(x)$ to allow all transactions to read all item-variants before making decision(s) - i.e. only an action-move without any consequences with regard to a CSG-graph.
Thus with reference to Table 9.6 in Section 9.4.1 this schedule corresponds to two single inconsistent retrieval situations. It is the "outer" transaction $T_1$ and the "inner" transaction $T_2$ that experience inconsistent retrievals.

From a cyclic total graph in Fig. 9.57 we have that

$$H_20 \not\in_c \text{Any Permutation of } T_1, T_2 \& T_3,$$

and thus we must deduce that:

$$H_20 \not\in \text{CSR}$$

But from the acyclic subgraph in Fig. 9.57 we also see that:

$$H_20 =_r T_3 \cup T_1 \cup T_2 =_r T_3 \cup T_2 \cup T_1$$

![Fig. 9.57. $A_p SG(H_m)$ vs. CSG($H_m$).](image)

And from Figs. 9.57a and 9.57b we see that:

$$H_20 =_x T_1 \cup T_3 \cup T_2$$

$$H_20 =_y T_2 \cup T_3 \cup T_1$$

Then we may conclude that:

$$H_20 \in A_R SR$$

Again if locking (even applying both shared- and exclusive-locks) is to be used as scheduling mechanism instead of the most general $A_p SG$-testing, the specification of transaction $T_1$ has to be changed into for example the following to make the resulting schedule which is shown, legal.

$T_1$:

Acquire One: Price $\leq 7.5$ $\&$ [Highest-Quality]
Still, as $H_{20}$ is not a member of CSR, the schedule could not of course be created by a 2PhaseLocking mechanism even applying both shared- and exclusive-locks. (Refer to Section 8.2.2).

An exclusion of either update $W_3(x)$ or update $W_3(y)$ from schedule $H_{20}$ would reflect a deletion of respectively the first or second part of the specification of transaction $T_3$. The resulting schedule would now correspond to one double dependent updates situation and one "indirect" inconsistent retrieval situation. Here the "indirect" coupling between two of the transactions consists of a double dependent updates situation instead of a write-write conflict as in situation C in Section 9.3.2. Refer also to the next example $E_4$*. Observe that it is not possible to create a valid example with one double dependent updates situation combined with one or more single inconsistent retrieval situation(s) on the same set of item-variants.
EXAMPLE $E_1^*$

Last, let us imagine the skeleton-database containing the following two item-variants.

\[
\begin{align*}
x: & \\
\text{Price} & = 10 \$ \\
\text{Quality} & = 95 \% \\
\text{No-Left} & = 3 \\
y: & \\
\text{Price} & = 5 \$ \\
\text{Quality} & = 90 \% \\
\text{No-Left} & = 1
\end{align*}
\]

We will now look at the following three wander-transactions.

\[
\begin{align*}
T_1: & \\
\text{Acquire One: Price} & \leq 15 \$ & \text{& Highest-Quality} \\
T_2: & \\
\text{Acquire One: Price} & \leq 15 \$ & \text{& Highest-Quality} \\
T_3: & \\
\text{Acquire One: Price} & \leq 15 \$ & \text{& Highest-Quality} \\
\text{& Acquire One: Lowest-Price} & & \text{& Quality} \geq 85 \%
\end{align*}
\]

A system schedule corresponding to the indicated wander-transactions accessing the given item-variants, is:

\[
H_{21}^* =
\begin{align*}
T_1: & \\
R_1(x)R_1(y)W_1(x) \\
T_2: & \\
R_2(x)R_2(y)W_2(x) \\
T_3: & \\
R_3(x)R_3(y)W_3(x) & \text{& } W_3(y)
\end{align*}
\]

We see that schedule $H_{21}$ resembles schedule $H_{15}$ from Section 9.4.2. Two changes are that retrieval $R_1(y)$ has been moved in between retrieval $R_1(x)$ and update $W_1(x)$, and that retrieval $R_2(y)$ has been moved in between retrieval $R_2(x)$ and update $W_2(x)$, both to allow all transactions to read all item-variants in the same sequence - i.e. only action-moves without any consequences with regard to a CSG-graph. Further retrieval $R_3(y)$ has been moved in between retrieval $R_3(x)$ and
update $W_3(x)$ to allow all transactions to read all item-variants before making decision(s) - i.e. again only an action-move without any consequences with regard to a CSG-graph.

Thus with reference to Table 9.6 in Section 9.4.1 this schedule corresponds to one single inconsistent retrieval situation and one "indirect" inconsistent retrieval situation. Here the "indirect" coupling between two of the transactions consists of a single inconsistent retrieval situation instead of a write-write conflict as in situation C in Section 9.3.2. It is the "inner" transactions $T_1$ and $T_2$ that experience respectively direct and indirect inconsistent retrievals.

From a cyclic total graph in Fig. 9.58 we have that

$$H_{21} \neq_h \text{Any Permutation of } T_1, T_2 \& T_3,$$

and thus we must deduce that:

$$H_{21} \notin \text{CSR}$$

But from the acyclic subgraph in Fig. 9.58 we also see that:

$$H_{21} \equiv_{r_a} T_3 \odot T_1 \odot T_2$$

![Fig. 9.58. $A_{r_a}SG(H_{21})$ vs. CSG(H_{21}).](image)

And from Figs. 9.58a and 9.58b we see that:

$$H_{21} \equiv_{x_c} T_3 \odot T_1 \odot T_2$$

$$H_{21} \equiv_{y_c} T_1 \odot T_2 \odot T_3 =_{y_c} T_2 \odot T_1 \odot T_3$$

Then we may conclude that:

$$H_{21} \in A_{r_a}SR$$
Again since $H_{21}$ is not a member of CSR, this schedule could not of course be created by a 2PhaseLocking mechanism even applying both shared- and exclusive-locks. (Refer to Section 8.2.2).

An exclusion of update $W_3(x)$ from schedule $H_{21}$ would reflect a deletion of the first part of the specification of transaction $T_3$. The resulting schedule would again correspond to one double dependent updates situation and one "indirect" inconsistent retrieval situation. Here the "indirect" coupling between two of the transactions consists of a double dependent updates situation instead of a write-write conflict as in situation C in Section 9.3.2. Observe once more that it is not possible to create a valid example with one double dependent updates situation combined with one or more single inconsistent retrieval situation(s) on the same set of item-variants.
9.5.2 **Specific Examples**

Now we continue with three different examples illustrating some (further) concurrency control and recovery aspects of a single transaction. With reference to Section 8.1.3 the transaction to be covered in the first example corresponds once more (like in the previous section) to true parallel behaviour. Thus it deals again with a departure from one-after-the-other atomicity. (See also Section 8.2.2). With reference to Section 8.1.4 the transactions to be covered in the second and third examples correspond to true partial behaviour — both between local databases and within a local database. Thus they deal with departures from both all-or-none unity/atomicity and all-or-nothing atomicity. (See also Section 8.2.2).

**Example 1**

First, let us imagine the skeleton-database containing the following three item-variants.

\[
\begin{array}{l}
\text{x:} \\
\begin{array}{ll}
\text{Price} & = 10 \text{ $} \\
\text{Quality} & = 95 \% \\
\text{No-Left} & = 1 \\
\end{array} \\
\end{array}
\]

\[
\begin{array}{l}
\text{y:} \\
\begin{array}{ll}
\text{Price} & = 5 \text{ $} \\
\text{Quality} & = 90 \% \\
\text{No-Left} & = 1 \\
\end{array} \\
\end{array}
\]

\[
\begin{array}{l}
\text{z:} \\
\begin{array}{ll}
\text{Price} & = 2.5 \text{ $} \\
\text{Quality} & = 87.5 \% \\
\text{No-Left} & = 1 \\
\end{array} \\
\end{array}
\]

As usual, all three item-variants belong to different local databases in our skeleton-database.
We will look at the following wander-transaction.

\[ T_1: \]
\[ \text{Acquire One: Lowest Price & Quality } \geq 85\% \]

Here we suppose that locking is used as scheduling mechanism (instead of the most general A_kSG-testing). Observe thus the original treatment of the 2PL-type notions in Section 2.2.7.

An initial part of a transaction schedule corresponding to the given wander-transaction in a hypothetical system schedule, is:

\[ T_1 = \]
\[ LS_1(x) \ R_1(x) \ LS_1(y) \ R_1(y) \ UL_1(x) \ LS_1(z) \ [R_1(z) \ UL_1(y)] \ldots \]

After \( T_1 \) has retrieved item-variant \( y - R_1(y) \) and found that \( y \) is a better offer than \( x \) (and the best so far), it may unlock item-variant \( x - UL_1(x) \). Further before \( T_1 \) may retrieve item-variant \( z - R_1(z) \), it has to lock item-variant \( z - LS_1(z) \). (And after having retrieved \( z \) and found \( z \) an even better offer than \( y \), \( T_1 \) may again unlock \( y \) etc.). Thus we have at least:

\[ UL_1(x) < LS_1(z) \]

This shows a definite break with the 2PhaseLocking rules. So for wander-transactions in skeleton-databases we do not have to enforce membership in the 2PL class even for dynamic syntax information only. (See also the discussion in Section 8.2.2).
EXAMPLE 2

Then, let us imagine the skeleton-database containing the following four item-variants.

\[
x: \\
\begin{array}{|l|}
\hline
\text{Price} = 10 \, \$ \\
\text{Quality} = 95 \, \% \\
\text{No-Left} = 1 \\
\hline
\end{array}
\]

\[
y: \\
\begin{array}{|l|}
\hline
\text{Price} = 5 \, \$ \\
\text{Quality} = 90 \, \% \\
\text{No-Left} = 1 \\
\hline
\end{array}
\]

\[
u: \\
\begin{array}{|l|}
\hline
\text{Price} = 7.5 \, \$ \\
\text{Quality} = 92.5 \, \% \\
\text{No-Left} = 1 \\
\hline
\end{array}
\]

\[
v: \\
\begin{array}{|l|}
\hline
\text{Price} = 2.5 \, \$ \\
\text{Quality} = 87.5 \, \% \\
\text{No-Left} = 1 \\
\hline
\end{array}
\]

This time still all four item-variants belong to different local databases in our skeleton-database.

We will again look at the following wander-transaction.

\[T_1: \text{ Acquire One: Lowest Price \& Quality } \geq 85 \, \% \]
Here we suppose that the two local databases owning item-variants u and v are either not willing to or not able to let transaction T₁ retrieve a specific item(-variant) and eventually later update the same item(-variant). This directly refers to 2PC-type problems as discussed in Section 3.1.4, and also indirectly to further 2PW-type problems as discussed in Section 3.1.6.

A transaction schedule corresponding to the given wander-transaction in a hypothetical system schedule, is:

\[ T₁ =  \\
R₁(x) R₁(y) . W₁(y) \]

After T₁ has retrieved item-variants x and y - R₁(x) and R₁(y), the transaction or the system knows or experiences that T₁ may not retrieve (and update) item-variants u and v. The access may be permanently impossible because T₁ lacks rights to do so - due to some (local) independent decision makers. The access may also be just temporarily impossible because T₁ lacks options to do so - due to some (global) partial failures. But our type of transaction-database pair allows just optimal treatment - i.e. eventually only going for the best available solution - and not only total treatment - i.e. always just going for the best possible solution. Thus T₁ may settle by acquiring the best item-variant among the available ones - without (waiting for) the possibility to check (all) the other unavailable item-variants. It updates item-variant y - W₁(y).

This shows a specific break with 2PhaseCommitment between cooperating autonomous nodes. So for wander-transactions in skeleton-databases we do not necessarily have to enforce all-or-none unity/atomicity. (See also the discussion in Section 8.2.2).

Observe that this kind of step-wise control of a transaction may be managed manually by the transaction or automatically by the system.
EXAMPLE 3

Last, let us imagine the skeleton-database containing the following four item-variants.

\[
\begin{array}{ll}
x_a: & x_b:
\begin{array}{ll}
\text{Price} & = 10 \ $ \\
\text{Quality} & = 95 \% \\
\text{No-Left} & = 1
\end{array} & \begin{array}{ll}
\text{Price} & = 7.5 \ $ \\
\text{Quality} & = 92.5 \% \\
\text{No-Left} & = 1
\end{array}
\end{array}
\]

\[
\begin{array}{ll}
y_a: & y_b:
\begin{array}{ll}
\text{Price} & = 5 \ $ \\
\text{Quality} & = 90 \% \\
\text{No-Left} & = 1
\end{array} & \begin{array}{ll}
\text{Price} & = 2.5 \ $ \\
\text{Quality} & = 87.5 \% \\
\text{No-Left} & = 1
\end{array}
\end{array}
\]

This time however two of the item-variants \(x_a\) and \(x_b\) belong to one specific local database "X" in our skeleton-database, while the other two item-variants \(y_a\) and \(y_b\) belong to another specific local database "Y" in our skeleton-database. (See also Eq. 9.26 and Fig. 9.7 in Section 9.1.2).

We will look at the following wander-transaction.

\[
T_1:
\begin{array}{l}
\text{Acquire One}_A: \text{Lowest Price} \ & \text{Quality} \geq 85 \% \\
\text{"X"}
\end{array}
\begin{array}{l}
\text{Acquire One}_B: \text{Price} \leq 15 \ $ \ & \text{Highest Quality}
\end{array}
\]

Here we suppose that the two item-types (A and B) that \(T_1\) basically wants to acquire together, are so that it may settle for only an A-type without any B-type, but it may not settle for only a B-type without any A-type. This effectively refers to 2PW-type problems as discussed in Section 3.1.6, and also additionally to further 2PC-type problems as discussed in Section 3.1.4.

A transaction schedule corresponding to the given wander-transaction in a hypothetical system schedule, is:

\[
T_1 = R_1(x_a) R_1(x_b) R_1(y_a) W_1(y_a) C_1(A) R_1(y_b) W_1(x_b) C_1(B)
\]

After \(T_1\) has retrieved item(-variant) s \(x_a\) and \(x_b\) at site "X" and item(-variant) \(y_a\) at site "Y" - \(R_1(x_a), R_1(x_b)\) and \(R_1(y_a)\), it knows which item-variant of A-type is the best, and thus it updates \(y_a\) -
W₁(yₐ). As a best A-type item-variant may be acquired separately, it even commits its access(es) referring to item-variants of A-type at site(s) "Y" (and "X") - C₁(A). (And this is done even though the best B-type item-variant later may be found to be the one at site "Y" too - again in accordance with the fact that our type of transaction-database pair allows just optimal treatment, and not only total treatment). After T₁ also has retrieved item(-variant) yₐ at site "Y" - R₁(yₐ), it further knows which item-variant of B-type is the best, and thus it updates xₐ - W₁(xₐ). As the acquisition of a best A-type item-variant already is committed, it only commits its access(es) referring to item-variants of B-type at site(s) "X" (and "Y") - C₁(B).

This shows a specific break with 2PhaseWriting within cooperating autonomous nodes. So for wander-transactions in skeleton-databases we do not necessarily have to enforce all-or-nothing atomicity. (See also the discussion in Section 8.2.2).

Observe that this kind of step-wise termination of a transaction must be controlled by the transaction itself.
9.5.3 Examples and Counter-Examples

Here we finish with two different examples illustrating that a wander-
transaction accessing a skeleton-database often needs to update an
item-variant before it retrieves another item-variant. If no write
could ever precede a read in any transaction schedule, the global part
of the membership requirements for class $A_{SR}$ would be assured for any
system schedule without any synchronization. Hence the binary relation
WR(Any Schedule) would be a partial order per definition. This is
similar to the effects of our criterion in the single-action-multi-
item model. See Section 9.2.2. The first example shows how the type of
access-predicate of a wander-transaction may lead to a write-precede-
read situation. While the second example shows how the type of item-
distribution in a skeleton-database may lead to a write-precede-read
situation. Observe that read-before-write refers to a single item-
variant, while write-precede-read refers to different item(-variant)s.

Example 1

First, let us imagine the skeleton-database containing the following
four item-variants.

x:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Price</td>
<td>10 $</td>
<td></td>
</tr>
<tr>
<td>Quality</td>
<td>95 %</td>
<td></td>
</tr>
<tr>
<td>No-Left</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

y:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Price</td>
<td>5 $</td>
<td></td>
</tr>
<tr>
<td>Quality</td>
<td>90 %</td>
<td></td>
</tr>
<tr>
<td>No-Left</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

u:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Price</td>
<td>7.5 $</td>
<td></td>
</tr>
<tr>
<td>Quality</td>
<td>92.5 %</td>
<td></td>
</tr>
<tr>
<td>No-Left</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

v:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Price</td>
<td>2.5 $</td>
<td></td>
</tr>
<tr>
<td>Quality</td>
<td>87.5 %</td>
<td></td>
</tr>
<tr>
<td>No-Left</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
As usual all four item-variants belong to different local databases in our skeleton-database.

We will look at the following three wander-transactions.

\[ T_{11}: \]
- Acquire One: Price \( \leq 5 \) $
- Acquire One: Price \( \leq 7.5 \) $ & Highest-Quality

\[ T_{12}: \]
- Acquire One: Price \( \leq 5 \)
- Acquire One: Price \( \leq 15 \) $ & Highest-Quality

\[ T_2: \]
- Acquire One: Lowest-Price & Quality \( \geq 85\% \)
- Acquire One: Price \( \leq 15 \) $ & Highest-Quality

Note that \( T_{11} \) and \( T_{12} \) are two transaction-variants only differing on the maximum price in the second part of their specification.

A condition as reflected in the first part of \( T_{11}/T_{12} \) is quite sensible. It may indicate that an article with such a low price is worth acquiring - irrespective of the quality.

A set of transaction schedules corresponding to the indicated wander-transactions accessing the given item-variants, are:

\[ T_{11} = R_1(x)R_1(y)W_1(y)R_1(u)R_1(v)W_1(u) \]

\[ T_{12} = R_1(x)R_1(y)W_1(y)R_1(u)R_1(v)W_1(x) \]

\[ T_2 = R_2(x)R_2(y)R_2(u)R_2(v)W_2(v)W_2(x) \]

Observe that \( T_{11}/T_{12} \) should acquire an article in accordance with the first part of the specification as soon as possible. Hence the transaction updates item-variant \( y \) immediately after it has been retrieved. Such a good offer ought to be acted upon at once.

If the transaction continued to search for an even better offer - i.e. corresponding to the same (or lower) price but a higher quality, another transaction might decide to acquire the specific article in the meantime. Thus a very good offer may get lost if not acted upon at once.
It is not a solution to use locking to "keep" a favourable offer available. Employing update- or exclusive-locks in such cases will limit the possible parallelism too much to be allowed. It will be counter-effective to the whole idea of wander-transactions accessing a skeleton-database. While employing shared-locks in these cases may very well lead to future deadlock situations in which the specific transaction might be selected for abortion.

Recall from Section 9.2.1 that our global criterion implies priority to transactions that make quick decisions. The sooner a transaction acquires articles, the higher priority it gets in its schedule.

To acquire an article is definitely optimal compared to not acquiring it at all. Updating any item-variant complying with a condition on attribute-values may thus be as optimal as updating the best item-variant among several item-variants complying with a condition on attribute-values.

Effectively a wander-transaction often needs to update an item-variant before it retrieves another item-variant. Here it is the type of access-predicate of a wander-transaction that leads to a write-precede-read situation. This makes the global part of the _A_5SR-criterion not superfluous. The following cases illustrate this in depth.

Now let us indicate which item-variants that the different wander-transactions would acquire if executed in isolation:

\[ T_{11}: y \land u \]
\[ T_{12}: y \land x \]
\[ T_2: v \land x \]

Hence combining \( T_{11} \) and \( T_2 \) induces no race-situation, while combining \( T_{12} \) and \( T_2 \) induces a race for item-variant \( x \).

Then let us look at four specific system schedules corresponding to the above given transaction schedules:

\[
H_{22} = T_{11}: R_1(x)R_1(y)W_1(y) \quad R_1(u)R_1(v)W_1(u) \\
T_2: R_2(x)R_2(y)R_2(u)R_2(v) \quad W_2(v)W_2(x)
\]

\[
H_{23} = T_{12}: R_1(x)R_1(y)W_1(y) \quad R_1(u)R_1(v)W_1(x) \\
T_2: R_2(x)R_2(y)R_2(u)R_2(v) \quad W_2(v)W_2(?)
\]

\[
H_{24} = T_{11}: R_1(x)R_1(y)W_1(y) \quad R_1(u)R_1(v)W_1(u) \\
T_2: R_2(x)R_2(y)R_2(u)R_2(v)W_2(v)W_2(x)
\]

\[
H_{25} = T_{12}: R_1(x)R_1(y)W_1(y) \quad R_1(u)R_1(v)W_1(?) \\
T_2: R_2(x)R_2(y)R_2(u)R_2(v)W_2(v)W_2(x)
\]
H_{22} observes both our global criterion (per database) and our local criterion (per item-variant). Thus it is a member of A_{SR} even though it is not a member of CSR. The acquired item-variants are as if both transactions were executed in isolation:

T_{11}: y & u 
T_{2}: v & x 

Note that a locking mechanism applying both shared- and exclusive-locks would allow the H_{22}-schedule unchanged.

H_{23} observes our global criterion. However it does not observe our local criterion if W_2(?) maps into W_2(x). The acquired item-variants would in this case be as if each transaction was executed in isolation:

T_{12}: y & x 
T_{2}: v & x 

But as there is only one instance left of the x-offer, this is not sensible. The corresponding single lost update situation makes the H_{23}-schedule neither a member of CSR nor a member of A_{SR}.

The write-read conflict on item-variant y (∋ T_{12} → T_{2}) having occurred, should give T_{12} priority over T_{2}. Hence a general solution to the non-observation of the local criterion is to make the system abort transaction T_{2} when T_{2} tries to update item-variant x. If a locking mechanism with both shared- and exclusive-locks is employed, a special solution is to make the system select transaction T_{2} for abortion when T_{12} and T_{2} both want to upgrade their shared-locks on item-variant x to exclusive-locks. After later rescheduling transaction T_{2}, the item-variants actually acquired would be:

T_{12}: y & x 
T_{2}: v & u 

This is the most sensible result.

H_{24} does not observe our global criterion. However it observes our local criterion. The acquired item-variants would in any case again be as if both transactions were executed in isolation:

T_{11}: y & u 
T_{2}: v & x 

Note that a locking mechanism applying both shared- and exclusive-locks would allow the H_{24}-schedule unchanged.

The inherent double inconsistent retrievals situation makes the H_{24}-schedule neither a member of CSR nor a member of A_{SR} However as the result is quite sensible, this schedule should perhaps be permitted. (But consult also the corresponding situation in Example 2). It is yet
another reminder of the fact that no correctness criteria can be tailor-maid to all and only all sensible situations.

Anyway the write-read conflict on item-variant y (→ T₁₁ → T₂) having first occurred, should give T₁₁ priority over T₂. Hence the system chooses to abort transaction T₂ when T₁₁ tries to retrieve item-variant v. The occurring write-read conflict on item-variant v (→ T₂ → T₁₁) closes a cycle in the A₉SG-graph. This approach effectively solves the non-observation of the global criterion. After later rescheduling transaction T₂, the item-variants actually acquired would be:

T₁₁: y & u
T₂: v & x

This is naturally the same result.

H₂₅ does not observe our global criterion. Further it does not observe our local criterion if W₁(?) maps into W₁(x). The acquired item-variants would in this case again be as if each transaction was executed in isolation:

T₁₂: y & x
T₂: v & x

Still as there is only one instance left of the x-offer, this is not sensible. The corresponding single lost update situation and the inherent double inconsistent retrievals situation make the H₂₅-schedule neither a member of CSR nor a member of A₉SR.

Solving the non-observation of the local criterion as we did for the H₂₃-schedule, might lead to an abortion of transaction T₁₂. After later rescheduling transaction T₁₂, the item-variants actually acquired would be:

T₁₂: y & u
T₂: v & x

However the write-read conflict on item-variant y (→ T₁₂ → T₂) having first occurred, should again give T₁₁ priority over T₂. Hence the system ought to abort transaction T₂ (instead of T₁₂) when T₁₂ tries to retrieve item-variant v. The occurring write-read conflict on item-variant v (→ T₂ → T₁₂) closes a cycle in the A₉SG-graph. Alternatively the system ought to select transaction T₂ (instead of T₁₂) for abortion when T₂ and T₁₂ both want to upgrade their shared-locks on item-variant x to exclusive-locks. Both these approaches effectively solve the non-observation of both the global and local criteria. After later rescheduling transaction T₂, the item-variants actually acquired would now be:

T₁₂: y & x
T₂: v & u

This is the most sensible result.
We have thus covered all the four combinations of observation/non-observation of the global/local criterion. On our way we have also given some corresponding examples and counter-examples with respect to membership in class $A_{SR}$. The main objective has been to illustrate that the global part of the $A_{SR}$-criterion is not superfluous. The examples and counter-examples have been too simple too show the full power of our correctness criteria.

(Observe that the last three examples employed in Section 9.5.1 may be converted into cases where at least one of the wander-transactions updates an item-variant before it retrieves another item-variant. Just delete the quality-reference in the first part of the last wander-transaction in each example; i.e. go from an access-predicate of the "old" type to an access-predicate of the "new" type as mentioned here. This corresponds exactly to a possible reversion of the action-moves introduced in those three examples to allow all wander-transactions to read all item-variants before making decisions).
EXAMPLE 2

Last, let us imagine the skeleton-database containing the following six item-variants.

\[ X_a: \begin{array}{l} \text{Price} = 15 \text{ $} \\ \text{Quality} = 100\% \\ \text{No-Left} = 2 \end{array} \]

\[ X_b: \begin{array}{l} \text{Price} = 7.5 \text{ $} \\ \text{Quality} = 92.5\% \\ \text{No-Left} = 2 \end{array} \]

\[ X_c: \begin{array}{l} \text{Price} = 12.5 \text{ $} \\ \text{Quality} = 97.5\% \\ \text{No-Left} = 1 \end{array} \]

\[ U_a: \begin{array}{l} \text{Price} = 5 \text{ $} \\ \text{Quality} = 90\% \\ \text{No-Left} = 1 \end{array} \]

\[ U_b: \begin{array}{l} \text{Price} = 10 \text{ $} \\ \text{Quality} = 95\% \\ \text{No-Left} = 1 \end{array} \]

\[ V_a: \begin{array}{l} \text{Price} = 2.5 \text{ $} \\ \text{Quality} = 87.5\% \\ \text{No-Left} = 2 \end{array} \]

This time however two of the item-variants \( y_a \) and \( y_b \) belong to one specific local database "Y" in our skeleton-database, and two other item-variants \( u_b \) and \( u_c \) belong to another specific local database "U" in our skeleton-database. Further there is an item-variant \( x_a \) which belongs to a third specific local database "X", and there is an item-variant \( v_c \) which belongs to a fourth specific local database. (Again see Eq. 9.26 and Fig. 9.7 in Section 9.1.2). Note that some item-variants (i.e. \( x_b, x_c, y_c, u_a, v_b \) and \( v_b \)) are lacking compared to the case where the local databases of each separate site have entries for exactly the same set of items. (See also Section 9.6.2).
We will look at the following two wander-transactions.

$T_1$:
- **Acquire One$_A$:** Lowest Price & Quality $\geq 85\%$
- **Acquire One$_B$:** Lowest Price & Quality $\geq 85\%$
- **Acquire One$_C$:** Price $\leq 15\$ & Highest Quality

$T_2$:
- **Acquire One$_A$:** Price $\leq 15\$ & Highest Quality
- **Acquire One$_B$:** Price $\leq 15\$ & Highest Quality
- **Acquire One$_C$:** Price $\leq 15\$ & Highest Quality

Note that the access-predicates of the wander-transactions are of the "old" type; i.e. the "new" type of access-predicate mentioned in Example 1 is not used.

A set of transaction schedules corresponding to the indicated wander-transactions accessing the given item-variants, are:

\[ T_1 = R_1(x_a)R_1(y_a)R_1(y_b)W_1(y_a)R_1(u_b)R_1(u_c)W_1(u_b)R_1(v_c)W_1(u_c) \]

\[ T_2 = R_2(x_a)R_2(y_a)R_2(y_b)W_2(x_a)R_2(u_b)R_2(u_c)W_2(y_b)R_2(v_c)W_2(u_c) \]

Observe that even $T_1/T_2$ should acquire an article in accordance with any part of the specification as soon as possible. The arguments are effectively the same as in Example 1. The best available offer ought to be acted upon as soon as this becomes apparent. Otherwise it may get lost; i.e. other transactions might decide to acquire the specific article instead.

Hence transaction $T_1$ updates item-variants $y_a$ and $u_b$ immediately after having retrieved all item-variants at respectively site "Y" and site "U". Likewise transaction $T_2$ updates item-variants $x_a$ and $y_b$ immediately after having retrieved all item-variants at respectively site "Y" and site "U". Remember that site "U" has no offer corresponding to article $A$, and that site "Y" has no offer corresponding to neither article $A$ nor article $B$.

Situations where the local databases of each separate site do not have entries for exactly the same set of items, would be quite normal.

Effectively a wander-transaction often needs to update an item-variant before it retrieves another item-variant. Here it is the type of item-distribution in a skeleton-database that leads to a write-precede-read situation. This again makes the global part of the $A_B$SR-criterion not superfluous. The following cases illustrate this in depth.
Now let us indicate which item-variants that the different wander-
transactions would acquire if executed in isolation:

\[ T_1: \, y_a, \, u_b \, \& \, u_c \]

\[ T_2: \, x_a, \, y_b \, \& \, u_c \]

Hence combining \( T_1 \) and \( T_2 \) induces a race for item-variant \( u_c \).

Then let us look at two specific system schedules corresponding to the
the above given transaction schedules:

\[
H_{26} = \\
T_1: \, R_1(x_a) \\\nT_2: \, R_2(x_a)R_2(y_a)R_2(y_b)W_2(x_a)R_2(u_b)R_2(u_c)W_2(y_b) \& \\\n\ldots \, R_1(y_a)R_1(y_b)W_1(y_a)R_1(u_b)R_1(u_c)W_1(u_b)R_1(v_c)W_1(v_c) \perp \\\n\ldots \, R_2(v_c)W_2(u_c) \perp \\
\]

\[
H_{27} = \\
T_1: \, R_1(x_a)R_1(y_a)R_1(y_b)W_1(y_a) \\\nT_2: \, R_2(x_a)R_2(y_a)R_2(y_b)W_2(x_a) \& \\\n\ldots \, R_1(u_b)R_1(u_c)W_1(u_b)R_1(v_c)W_1(v_c) \perp \\\n\ldots \, R_2(u_b)R_2(u_c)W_2(y_b)R_2(v_c)W_2(u_c) \perp \\
\]

We have to break the system schedules each into two parts due to their
length. The lower part is supposed to follow the upper part.

In Section 9.6.2 we will generalize the local criterion from one item
per site to several items per site. In the normal case this implies
going from conflict serializability per item-variant to conflict serializability per site.

\( H_{26} \) observes both our global criterion (per database) and the
generalized local criterion (per site). Thus it is a member of \( ApSR \)
even though it is not a member of CSR. The item-variants actually
acquired are:

\[ T_1: \, y_a, \, u_b \, \& \, v_c \]

\[ T_2: \, x_a, \, y_b \, \& \, u_c \]

For a locking mechanism with both shared- and exclusive-locks to allow
the \( H_{26} \)-schedule, the retrieval \( R_1(y_a) \) has to be moved before the
update \( W_2(x_a) \). Such an action-move has no consequences with regard to
a CSG-graph.

It is quite sensible that item-variant \( u_c \) is acquired by transaction
\( T_2 \) instead of transaction \( T_1 \). The write-read conflicts on item-
variants \( y_b \) and \( u_c \) (\( \Rightarrow \, T_2 \, \Rightarrow \, T_1 \)) having occurred, should give \( T_2 \)
priority over \( T_1 \).
**H₂₇** does not observe our global criterion. However it observes the generalized local criterion. The acquired item-variants would in this case again be:

\[ T₁: yₐ, uₐ & vₐ \]

\[ T₂: xₐ, yₐ & uₐ \]

For a locking mechanism with both shared- and exclusive-locks to allow the H₂₇-schedule, the retrieval \( R₁(uₐ) \) has to be moved before the update \( W₂(yₐ) \). Such an action-move has no consequences with regard to a CSG-graph.

The inherent double inconsistent retrievals situation makes the H₂₇-schedule neither a member of CSR nor a member of AₚSR. Actually it is now not sensible that item-variant \( uₐ \) is acquired by transaction \( T₂ \) instead of transaction \( T₁ \). The write-read conflict on item-variant \( yₐ \) (\( T₁ → T₂ \)) having occurred before the write-read conflict on item-variant \( uₐ \) (\( T₂ → T₁ \)), should give \( T₁ \) priority over \( T₂ \).

Hence the system ought to abort transaction \( T₂ \) when \( T₁ \) tries to retrieve item-variant \( uₐ \). The occurring write-read conflict on item-variant \( uₐ \) (\( T₂ → T₁ \)) closes a cycle in the AₚSG-graph. This approach effectively solves the non-observation of the global criterion. After later rescheduling transaction \( T₂ \), the item-variants actually acquired would then be:

\[ T₁: yₐ, uₐ & uₐ \]

\[ T₂: xₐ, yₐ & vₐ \]

This is the most sensible result.

We have thus covered the two basic combinations of observation/non-observation of the global criterion combined with observation of the generalized local criterion. The two remaining combinations of observation/non-observation of the global criterion combined with non-observation of the generalized local criterion are also easy to cover. The main objective has again been to illustrate that the global part of the AₚSR-criterion is not superfluous.

Observe that the last example employed in Section 9.5.2 also illustrated a wander-transaction that wanted to update an item-variant before it retrieved another item-variant. This even occurred in a situation where the local databases of each separate site have entries for exactly the same set of items.
9.6 CONTROL OF SYSTEMS

In this section we will discuss certain aspects of the implementation of wander-transactions in a skeleton-database. Thus we shall again only deal with our read-before-write case.

9.6.1 FROM LOGICAL CRITERION TO SYSTEM MECHANISM

We start with mappings from the requirements of our class \( A_{\text{SR}} \) - as specified in Section 9.2.1 - to applicable concurrency control mechanisms.

From Eq. 9.44 (and Eq. 9.41) in Section 9.2.1 we have for our local and global criteria:

\[
\forall x \in D \; [[\text{WR-}\text{RW}[\text{-WW}]_x](\text{Any Schedule}) \; \text{Partial Order}] \quad (\text{Eq. 9.92})
\]

\&

\[
\text{WR}(\text{Any Schedule}) \; \text{Partial Order} \quad (\text{Eq. 9.93})
\]

Thus both per single item (or item-variant to be precise) and per multiple item (or item-variant) it would be most natural and direct to employ a concurrency control mechanism derived from the SerializationGraphTesting (SGT) of Section 2.3.2. See also Section 3.2.1 for its distributed counter-part; i.e. Distributed-SGT (D-SGT). This is reflected in respectively Eqs. 9.12 to 9.13 in Section 9.1.1 or Eqs. 9.51 to 9.52 in Section 9.2.1 (for the local mechanism) and Eqs. 9.59 to 9.60 in Section 9.2.1 (for the global mechanism). These two mechanisms could be referred to as SingleItem-SGT (SI-SGT) and MultipleItems-"SGT" (MI-"SGT").

But note that we might actually derive our concurrency control mechanisms from any of the four basic mechanisms of Section 2.3.2. See also Section 3.2.1 for their distributed counter-parts. Hence it is possible to end up with either SingleItem-2PL, -TO, -SGT or -C and with either MultipleItems-"2PL", -"TO", -"SGT" or -"C". (We use hyphens to stress the point that the underlying requirements are weaker than those of the original corresponding mechanism).

Further with reference to the option to dissect concurrency control on different geographic locations as mentioned in Section 3.2.1, it should be possible to separate concurrency control for distinct items (or item-variants) from each other - i.e. synchronize accesses to one item separately and differently from accesses to another item. Any of the four basic mechanisms may be used as a local mechanism for each of the different items. Here of course we do not need any arrangement to make the resulting local serialization orders consistent - besides what is to be induced by the global mechanism itself.

And with reference to the option to dissect concurrency control on different types of conflicting operations as mentioned in Section 2.3.2 - i.e. to synchronize reads vs. writes (& writes vs. reads) separately and differently from writes vs. writes, it should be interesting to try to adapt this fact and its resulting techniques
(see especially [Bern80a]) to the separate but still associated local and global synchronization. Again any of the four basic mechanisms may be used as the global mechanism - eventually different from any local mechanism(s).

Let us now refer back to the discussion in Section 9.2.1 concerning the use of the binary relation WR as a priority mechanism.

We claimed that the high dominance of reads over writes and the high dominance of late writes over early writes inherent in the semantics of wander-transactions, make a global requirement which corresponds to the binary relation WR being a partial order, even suitable as a priority mechanism in controlling actual race-situations for favourable offers.

When a break with the local requirement per item-variant and/or per site is about to occur, the AₖSG-graph is consulted. If some of the involved wander-transactions (two or more) are ordered, there is at least a

\[ T_i \rightarrow \ldots \rightarrow T_k, T_k \rightarrow T_j \text{ Path} \]

among those transactions. Then wander-transaction \( T_i \) may be selected for abortion. Note the above quoted characteristics of wander-transactions. These make the read of \( T_i \) which has lead to the last arc together with a write of \( T_k \), with high probability the last read to have occurred among those reads contributing to the write-read conflicts reflected in the path. Hence the \( (T_i, T_j) \)-path mirrors a logical reads-from chain that also reflects with high probability a physical action-occurrence sequence. If the path contains two transactions only, this statement is true even per definition.

When a break with the global requirement per database is about to occur, the AₖSG-graph is consulted. We are about to insert a

\[ T_j \rightarrow T_i \text{ Arc,} \]

and there is already at least a

\[ T_i \rightarrow \ldots \rightarrow T_j \text{ Path.} \]

Then wander-transaction \( T_i \) is selected for abortion. Again note the above quoted characteristics of wander-transactions. These make the write of \( T_i \) which leads to the new arc together with a read of \( T_i \), with high probability the last write to have occurred among those writes contributing to the write-read conflicts reflected in the cycle about to be formed. Hence the \( (T_i, T_i) \)-cycle mirrors a logical reads-from chain that also reflects with high probability a physical action-occurrence sequence. If the old path contains two transactions only, this statement is true even with higher probability.

A high-accuracy priority mechanism would involve a representation of the single write- and read-actions and some corresponding timestamps in the AₖSG-graph. This implies nodes that are actions instead of transactions. Further it requires the introduction of node-labels that are action-timestamps. (Compare this with the transaction-timestamps used in [Rose78] and mentioned in Section 2.3.2). This high-overhead solution allows us to abort the transaction corresponding to the
actually youngest read/write (and not only the probably latest read/write) in connection with breaks with the local/global requirement.

The committing of a wander-transaction may lead to an immediate deletion of its eventually corresponding node in the $A_r$SG-graph. Committing transaction $T_k$ should force each eventually existing $T_i \rightarrow T_k$, $T_k \rightarrow T_j$ Path to be transformed into a $T_i \rightarrow T_j$ Arc.

This reflects an inheritance of reads-from relationships.

(The abortion of a wander-transaction will have similar effects. However the consequences will be more complex in this case).

The inheritance of reads-from relationships must also apply for an occurring write-read conflict where the writing transaction has already committed (/aborted).

To allow partial committing of wander-transactions as in Example 3 of Section 9.5.2, also induces partial abortion and partial rescheduling of transactions. The effects on the $A_r$SG-graph would be similar to the needs for action-representation instead of transaction-representation as mentioned above.

Is it sensible for a transaction to immediately acquire any article (implying writing) to gain priority in a schedule? The answer is negative. If you are not aborted, you have to pay for the purchase. So it ought to be an article that is really wanted. And to gain priority, it even has to be an article that others are interested in (implying reading). Further is it sensible for a wander-transaction to partially commit any write to avoid that it gets aborted? The answer is negative. If you are partially aborted, you have to pay for all the committed purchases. Some of these might not be actually wanted without in connection with some other purchases.

Let us also initiate - as an example - the discussion of how 2PhaseLocking (2PL) could be adapted to wander-transactions in a skeleton-database. See also the comments to the examples of Section 9.5.

First, we do not consider the generalized local criterion per site - as mentioned in Example 2 of Section 9.5.3.

- Before a transaction may check an item - by retrieving its value, it has to request an S-lock.

- And if the transaction decides not to acquire a specific item - by not updating its value, it may release the S-lock as soon as this fact becomes apparent. This may happen immediately if the current offer is not in accordance with the transaction-predicate at all - or if it is worse than a
previous offer. Alternatively this may not happen until later when a better offer is found.

This allows breaks with the original 2PL-rules for all items globally, as elaborated in Example 1 of Section 9.5.2.

- But if the transaction decides to acquire a specific item - by updating its value, it has to upgrade the S-lock to an X-lock.

This means that the transaction may not first release the S-lock and then later request an X-lock. Hence it leads to observation of the original 2PL-rules for each item locally.

- After the transaction has acquired a specific item - by updating its value, it may release the X-lock. But this must happen at a global locked-point or at global end. Partial commitment may though happen earlier or later - as elaborated in Example 3 of Section 9.5.2.

The adapted 2PL-rules that we have outlined as a starting point, correspond to a restricted 2PL- or LUE-variant of the C₂-class from Section 5.3. The items which are both retrieved and updated have to use the Upgrade Lock alternative. While the items which are only retrieved have to observe a "chained-locking"; i.e. the currently best offer has to be kept locked until a better offer is eventually found. This is too restrictive. Many situations actually allowed to occur with Adapted-SGT will not be allowed to occur with Adapted-2PL. The comments in Section 9.5 about the allowance or non-allowance of system schedules by a locking mechanism concerned the chained-locking only. Adding the 2PL- or LUE-type requirements on the updated items makes even fewer schedules allowed. This is just natural as 2PL is a stronger mechanism than SGT. See Sections 2.2.7 and 4.3.

Second, we do consider the generalized local criterion per site. See the next section.

The adapted 2PL-rules that we will outline as a starting point, correspond to a further restriction of the 2PL- or LUE-variant of the C₂-class given above. Again the X-locks have to be held until a global locked-point or until global end. While the S-locks not upgraded to X-locks have to observe the "chained-locking" - and have to be held until a local locked-point or until local end. This is once more too restrictive. Many situations actually allowed to occur with Adapted-SGT will not be allowed to occur with Adapted-2PL. This is partially related to the chained-locking, and partially related to the added 2PL- or LUE-type requirements locally and/or globally. The comments in Section 9.5 once more concerned the chained-locking only.

Observe that omitting all read-only actions (i.e. initial retrievals of specific items that are not accompanied by final updates on the same items) from any schedule observing the adapted 2PL-rules makes it even observe the original 2PL-rules. This applies to both the above cases. (Compare this with the fact from Section 9.4.2 effectively stating that omitting all read-only actions from any schedule being allowed by Adapted-SGT makes it even being allowed by Original-SGT).
Let us even indicate - as an example - that adapting 2PhaseLocking (2PL) to wander-transactions in a skeleton-database actually opens the way for some additional freedom with respect to achieving optimal solutions, and not only total solutions - see Section 8.1.4.

Basically, when an item to be checked by a transaction (through retrieving its value) is locked by another transaction, the first transaction could actually either be made to wait until this very item is freed for lock-setting (normal case) or be allowed to try whether the next following item - according to an initial ordering fixed by that transaction - is free for lock-setting (extraordinary case).

Further, when an item to be acquired by a transaction (through updating its value) is locked by another transaction, the first transaction could actually either be made to wait until this very item is freed for lock-upgrading (normal case) or be allowed to try whether the next best item - according to an eventual ranking made by that transaction - is free for lock-upgrading (extraordinary case). This last option of course requires that a transaction does keep more - by not releasing the locks - than the currently best offer of some kind.

Even, if an item being checked or acquired by a transaction does involve the transaction in a deadlock, options would exist similar to those quoted for the two above situations.
9.6.2 From Single Item per Site to Multiple Items per Site

We continue with a discussion of the combined inter-site and intra-site problem - see Eq. 9.26 and Fig. 9.7 in Section 9.1.2.

So far we have basically only treated the inter-site problem and skipped the intra-site problem. As indicated in Section 9.1.2 this has been done to focus on the main ideas. (However see Example 3 in Section 9.5.2 and Example 2 in Section 9.5.3 for illustrations and discussions of some related ideas).

Let us now refer back to the comments concerning Eq. 9.26. According to the new general view - i.e. where we operate even with several specific items per site, instead of the new special view - i.e. where we operate only with one unique item per site, \( x_a, x_b \) and \( x_c \) represent different items existing at a unique site, while \( x, y \) and \( z \) represent a unique item existing at different sites.

Thus with reference to Eq. 9.26, we will initially give the items belonging to a unique site a common name:

\[
\begin{align*}
X &= \{x_a, x_b, x_c, \ldots\} \\
Y &= \{y_a, y_b, y_c, \ldots\} \\
Z &= \{z_a, z_b, z_c, \ldots\}
\end{align*}
\]

(Eq. 9.94)

So with reference to Eq. 2.1 in Section 2.2.4, we have:

\[
D = X \cup Y \cup Z \cup \ldots
\]

Note that we must not necessarily anticipate that the local databases of each separate site (department store) have entries for exactly the same set of items (articles), as we did in Section 6.2. However the databases of each site must naturally have entries corresponding to partially overlapping subsets of a common set of items. If all subsets were totally non-overlapping, there would just be one "variant" of each item.

With reference to Eq. 9.7 (and Eq. 9.20) in Section 9.1.1, we further need to define one compound binary relation per unique site:

\[
\begin{align*}
\text{WR-RW-WW}_X(H) &= \bigcup_{x \in X} \text{WR-RW-WW}_x(H) \\
\text{WR-RW-WW}(H) &= \bigcup_{X \subseteq D} \text{WR-RW-WW}_X(H)
\end{align*}
\]

(Eq. 9.95)

Observe that we have such a binary relation for each separate site - \( X \subseteq D \) - in the system.
Finally we are able to state the correctness criteria for our general system.

Effectively, if the items of each site are not related by any integrity constraints - an extraordinary case, the requirements may be the same as those for our special system treated so far. Thus we may still employ the local and global criteria of Eqs. 9.92 and 9.93 in the previous section.

But, when the items of each site are related by some integrity constraints - the normal case, the requirements have to reflect this. As usual this implies going from $x$-conflict serializability per single item - $x \in D$ - to $X$-conflict serializability per unique site - $X \subset D$ (see Section 9.1.1). Thus we end up with the following local and global criteria:

\[
\forall X \subset D \ [\text{WR-RW[-WW]}_X(\text{Any Schedule}) \text{ Partial Order}] \quad \text{(Eq. 9.96)}.
\]

\[
\&
\]

\[
\text{WR(Any Schedule) Partial Order} \quad \text{(Eq. 9.97)}
\]

Notice the difference between Eq. 9.92 and Eq. 9.96. The changes show only locally. With reference to Eq. 9.41 in Section 9.2.1 the corresponding class is

\[
C_{G:WR} \cap C_{L:WR-RW-WW}
\]

where $L$ indicates per site instead of per single item.

We may now relate our correctness criteria to the fragmentwise serializability concept of [Garc88]. This notion is biased towards an increased availability in a replicated database (see Section 3.1.3). Hence it is not directly comparable. Basically fragmentwise serializability corresponds to the local criterion alone without the added global criterion; i.e. to Eq. 9.96 alone without the added Eq. 9.97. With reference to Eq. 9.29 in Section 9.1.2 the "corresponding" class is

\[
C_{L:WR-RW-WW}
\]

where $L$ indicates per fragment instead of per single item. Each fragment may here encompass the items in the local database(s) of one or more sites. The result is site serializability (one site) or region serializability (several sites) without any further global restrictions.

By the way, such fragmentwise serializability would correspond to the predicated serializability of Section 8.1.5 whenever the associated fragments could match the independent parts mentioned in the same section. Hence site or region serializability might be an appropriate correctness criterion in traditional distributed databases with predicated integrity constraints of the intra-fragment type only and not of both the intra-fragment and inter-fragment types.
9.6.3 FROM SINGLE LEVEL CRITERION TO MULTIPLE LEVELS CRITERIA

We finish with a discussion of the cofunctioning of alternative and traditional criteria - see Fig. 8.3 in Section 8.3.

Let us now repeat the traditional criterion - i.e. conflict serializability - from Section 9.1.1:

- WR-RW[-WW](Any Schedule) Partial Order (Eq. 9.98)

Of course this requirement refers to Eq. 9.20.

With reference to the different points of view defined in Section 8.3, the (combined) requirements of one or more global buyer(s) and one or more local owner(s) correspond exactly to the criteria of Eqs. 9.96 and 9.97 in the previous section, while the requirements of one or more global supplier(s) correspond directly to the criterion of Eq. 9.98.

Again we recall the fact from Section 9.4.2 stating that omitting all read-only actions (i.e. initial retrievals of specific items that are not accompanied by final updates on the same items) from any schedule of wander-transactions makes it serializable. Such an omission even does not change the resulting effects due to the semantics of wander-transactions accessing a skeleton-database.

From this it is possible to conclude that global buyers plus local owners and global suppliers may coexist without any problems. The system only has to synchronize each group of transactions

\[ G_1 = \text{Global Buyers} + \text{Local Owners} \]

\&

\[ G_2 = \text{Global Suppliers} \]

internally according to the requirements of that group:

\[ G_1: \text{Eqs. 9.96 - 9.97} \]

\[ G_2: \text{Eq. 9.98} \]

Thus the cofunctioning of these different points of view do not introduce any extra synchronization needs between the groups - besides those within the groups.
10 Further Steps

This last chapter points out our ideas about how to extend the theoretical and practical results gained so far.
10.1 **Needs for Further Revision of Assumptions and Results**

Let us once more review the main conclusions of two of the special paragraphs of Section 9.2.2 - further evaluating the results from the discussion and specifications of classes $A_{RS}$ and $A_{MR}$SR. Let us also review one main suggestion from Section 9.6.1 - initially discussing some system mechanisms corresponding to the logical criteria of class $A_{RS}$SR.

10.1.1 **Case Grouping**

Our main emphasis has been on the special read-before-write case and its corresponding correctness criterion. But we have also put some emphasis on an accompanied treatment of the general not-read-before-write case and its possible correctness criterion. And we have done this even without yet being able to state the exact application area of the results or the eventual combination of the results with those of the main case in its corresponding application area; i.e. wander-transactions in a skeleton-database.

An exploration of the open ends referred to above would certainly be interesting.

(Let us just mention our correctness criteria for a general system with multiple items per site even for the not-read-before-write case.

If the items of each site are not related by any integrity constraints - an extraordinary case, the requirements may be the same as those for our special system treated so far. Thus from Eq. 9.48 (and Eq. 9.45) in Section 9.2.1 we have:

$$\forall x \in D \left[ [WR-]RW-WW_x \text{(Any Schedule)} \text{ Partial Order} \right] \quad \text{(Eq. 10.1)}$$

$$\&$$

$$\text{WR-RW(Any Schedule) Partial Order} \quad \text{(Eq. 10.2)}$$

Compare this with Eqs. 9.92 to 9.93 in Section 9.6.1.

When the items of each site are related by some integrity constraints - the normal case, the requirements have to reflect this. Thus we end up with:

$$\forall X \subset D \left[ WR-RW-WW_{X} \text{(Any Schedule)} \text{ Partial Order} \right] \quad \text{(Eq. 10.3)}$$

$$\&$$

$$\text{WR-RW(Any Schedule) Partial Order} \quad \text{(Eq. 10.4)}$$

Compare this with Eqs. 9.96 to 9.97 in Section 9.6.2. Further notice the difference between Eq. 10.1 and Eq. 10.3).
10.1.2 Criteria Basis

CSR is formally not the only possible starting point for our new correctness criteria - corresponding to $A_{\text{CSR}}$ and $A_{\text{NRCSR}}$ - in absolutely all cases. But to have combinable and comparable criteria, CSR is the natural choice.

(Practically we also have to take the efficiency aspects into consideration. As mentioned in Section 4.3 this may rule out anything but CSR as criteria basis even for the remaining cases where there still are theoretically other possibilities).

An investigation of the extra cases referred to above would be interesting.

(However note that while FSR is definitely not a subclass of $A_{\text{NRCSR}}$, FSR is a subclass of $A_{\text{CSR}}$ for the read-before-write case considering that there are no dead retrieval actions - actually making FSR equal to CSR. See Eqs. 9.78a and 9.72a-b in Section 9.4.1).

10.1.3 System Mechanism

We might actually derive our concurrency control mechanisms from any of the four basic mechanisms of Section 2.3.2. See also Section 3.2.1 for their distributed counter-parts. Hence it is possible to end up with either SingleItem-2PL, -TO, -SGT or -C and with either MultipleItems-"2PL", -"TO", -"SGT" or -"C". (We use hyphens to stress the point that the underlying requirements are weaker than those of the original corresponding mechanism).

An elaboration of the inherent options referred to above would also be interesting.
10.2 Possibilities for Some More Expanded Systems

Now we will propose some extensions to some main points treated earlier.

Actually, the needs of separate wander-transactions in a skeleton-database may be different.

First, a simple way to indicate such specific wishes or requirements of a given wander-transaction is to attach one or more corresponding parameters (set by the user and observed by the system).

Some interesting issues that could be controlled through parameter values are:

- Which sites should be checked and which not for each specific item-type?

This deals with a possible trade-off referring to the cost of buying an article at a department-store vs. the cost of transporting the article from the department-store to the user. An article-type needed urgently, an article-type whose price and quality do not vary much throughout, or an article-type being extremely large or heavy could make certain department-stores located far away irrelevant to check. (See the reference to global access optimization and global data transparency assurance in Section 8.2.1).

Thus it concerns once more some additional possible freedom with respect to achieving optimal solutions, and not only total solutions.

- What should be done if a site to be checked for a specific item-type, is either unwilling or unable to honour the request for its corresponding item-variant?

This was treated in Example 2 of Section 9.5.2 as an example of the fact that we do not necessarily have to enforce all-or-none unity/atomicity for wander-transactions in a skeleton-database.

(We indicated already in Section 9.5.2 that an eventual associated step-wise control of the transaction might be managed manually by the transaction or automatically by the system).

- What should be done if several items to be acquired at a given site, do not have to be acquired together?

This was treated in Example 3 of Section 9.5.2 as an example of the fact that we do not necessarily have to enforce all-or-nothing atomicity for wander-transactions in a skeleton-database.
(We indicated already in Section 9.5.2 that an eventual associated step-wise termination of the transaction must be controlled by the transaction itself).

- What should be done if one item to be checked or acquired at a given site, does not allow a lock to be set or upgraded - or if one item being checked or acquired at a given site, gets involved in a deadlock?

This was treated in Section 9.6.1, and it concerns once more some additional possible freedom with respect to achieving optimal solutions, and not only total solutions.

These issues simply concern alternative ways of implementing (all) wander-transactions. Here we suggest that each specific transaction can be implemented differently through parameter settings according to the wishes of the corresponding user.

Another important aspect that might be controlled through a parameter value is:

- How should the transaction be synchronized against other transactions?

Thus this concerns what kind of view to achieve.

This aspect actually concerns alternative ways of specifying wander-transactions. We have defined a sensible criterion (see the discussions in Sections 9.2.1, 9.3.1 and 9.4.2) to be applied to all wander-transactions. Here we indicate that each specific transaction may be specified differently through parameter settings according to the requirements of the corresponding user.

Such an approach is open to effects like the different levels of non-serializability of Section 5.3. A pragmatic solution will again be for the system (effectively like in [Gray76]) to offer a certain set of transaction-profiles, among which any user has to choose one. This use of tailored transaction-classes also resembles the approach taken in the SDD-1 system (see Section 3.4).

One of these transaction-types may have requirements like the criterion we have defined, while some others may have requirements which are stronger than ours or even weaker than ours. Remember the argument from Section 9.2.1 that the lack of integrity-constraints in a skeleton-database actually leaves the determination of criteria for wander-transactions to choice.

Second, an advanced way to indicate such specific wishes or requirements of a given wander-transaction is to employ a dialogue form (between the user and the system).

Interesting issues and aspects possibly being supervised and acted upon through a dialogue are the same as those given above (both groups) for the static employment of parameter values.
But also the current results (from the traversing carried out so far in the skeleton-database) of any wander-transaction are open for being supervised and acted upon through a dynamic employment of a dialogue. Thus the user could at any moment during the execution be able to decide whether what is currently achieved, is a good enough solution or not. Hence a transaction might possibly not only give up a total solution, but even an optimal solution - again see Section 8.1.4.

And a further elaboration of this idea would be to allow the user to change the goal of the wander-transaction at any moment during the execution (in addition to allow him or her to decide whether the goal is reached or not). However such adaptable wander-transactions seem to introduce too much flexibility with respect to achieving a manageable system.
Appendix

In this appendix we are concerned with some specific material related to Sections 4.3 and 9.4.1.

On our way we have to present a definition of class FSR from Section 4.2.3 for the general multi-action-single-item model used in Section 2.2.6. We will continuously relate the specifications to those concerning the widest class so far from Section 2.2.6; i.e. VSR_{0}.

Like before we will make use of the extended version of a system schedule, so refer to the definitions of H^*, T_{w} and T_{r} in Eqs. 2.29 to 2.31. (Observe also the corresponding Eqs. 4.14 to 4.16).

The reads-x-from notion will here relate actions instead of transactions. Besides that, the specification equals that of Eqs. 2.18 to 2.20. (See also Eqs. 4.17 to 4.18).

R_{i} reads-x-from W_{j} in H iff

- W_{j}(x) < R_{i}(x) \hspace{1cm} \text{(Eq. A.1)}
- A_{j} \not< R_{i}(x) \hspace{1cm} \text{(Eq. A.2)}

and

- \forall W_{k}(x) \in h [W_{j}(x) < W_{k}(x) < R_{i}(x) \Rightarrow A_{k} < R_{i}(x)] \hspace{1cm} \text{(Eq. A.3)}

An illustration is given in Fig. A.1. Relate this to Fig. 4.4 (and Fig. 2.25).

![Figure A.1](image)

**Fig. A.1.** Definition of R_{i} Reads-x-From W_{j} in H^* with resulting contribution to FSG(H^*).

Also the liveness concept must here be associated with actions instead of transactions. The discussion below basically concerns exactly this point. A computation is shown in Fig. A.2. Relate this to Fig. 4.5.
Like before the schedule equivalence definition will employ the alternative approach; i.e. without writes-x-finally. Note again the dependence on actions instead of transactions. Compare the specification with Eqs. 2.32 to 2.33 and Eqs. 4.19 to 4.21.

**Final-state equivalent schedules** is a Binary Relation $=_{r}$:

$$
H =_{r} H' \text{ iff}
$$

- $h = h'$ \hspace{1cm} (Eq. A.4)
- $\text{live-actions}(H^*) = \text{live-actions}(H'^*)$ \hspace{1cm} (Eq. A.5)
- $\forall x \in D, R_i, W_j$ \hspace{1cm} (Eq. A.6)

$$
[[\text{Live } R_i \text{ reads-}x\text{-from } W_j \text{ in } H^* \wedge A_i \notin h \wedge A_j \notin h] \rightarrow \text{Live } R_i \text{ reads-}x\text{-from } W_j \text{ in } H'^*]
$$

The schedule serializability definition fully equals that of Eq. 4.22. (See also Eq. 2.28).

**Final-state serializable schedules** is a Set FSR:

$$
H \in \text{FSR} \text{ iff}
$$

- $\exists CH \in S [H =_{r} CH]$ \hspace{1cm} (Eq. A.7)

Hence FSR (like VSR$_0$) has complete system schedules only as members.

The schedule serializability test again depends on actions and not transactions. Compare the construct with Eqs. 2.34 to 2.35 and Eqs. 4.23 to 4.24.

- $\forall (FSG(H^*)) = \{W_w(x)|x \in D\} \cup \{R_r(x)|x \in D\}$ \hspace{1cm} (Eq. A.8)
- $\cup \{p|p \in h\}$
- A ( FSG(H*) ) =

\{ W_{w}(x) \rightarrow p | x \in D \land p \in h \} \quad \text{(a)}

\cup \{ p \rightarrow R_{r}(x) | x \in D \land p \in h \} \quad \text{(b)}

\cup \{ R_{j}(y) \rightarrow W_{j}(x) | R_{j}(y) \prec_{h} \text{Live } W_{j}(x) \} \quad \text{(c)}

\cup \{ W_{j}(x) \rightarrow R_{i}(x) \} \quad \text{(d)}

\text{Live } R_{i} \text{ reads } -x- \text{from } W_{j} \text{ in } H^{*} \}

\cup \{ W_{k}(x) \rightarrow W_{j}(x) \lor R_{i}(x) \rightarrow W_{k}(x) \} \quad \text{(e)}

\text{Live } R_{i} \text{ reads } -x- \text{from } W_{j} \text{ in } H^{*} \land \text{W}_{k} \text{ writes } -x- \text{also in } H^{*} \}

Once more: \( H \in \text{FSR} \iff \exists \text{ Directed Acyclic Graph } \text{FSG}_{a}(H^{*}) \in \text{Directed Graph-Set } \text{FSG}(H^{*}) \).

Case d) was basically illustrated in Fig. A.1. But observe that only \( x_{w}^{-} \)-arcs corresponding to live \( R_{j}(x) \)'s - as computed in Fig. A.2 - are to be included. \( x_{w}^{-} \)-arcs corresponding to dead \( R_{i}(x) \)'s are to be excluded. This is shown in part -) of Fig. A.3. Relate this to Fig. 4.6. Part +) of Fig. A.3 - corresponding to case c) - deals with the liveness relations within a single transaction. This is also to be discussed below. Case e) - concerning \( x_{w}^{-} \)-arc-pairs - is thoroughly illustrated in Fig. A.4. Relate this to Fig. 4.7 (and Fig. 2.26). Cases a) and b) respectively concern the initialization and checking of the database.

(From this it is possible to formally deduce that FSR \( \supset \text{VSR}_{0} \), even in the general model).

We have employed an action-graph instead of the transaction-graphs normally used to pinpoint the essential notions. In a real situation
Fig. A.4. "Definition" of \( \text{Write-} x\text{-Also in } H^* \)
for \( R_i(x) \text{ Live and } R_i \text{ Reads-} x\text{-from } W_j \text{ in } H^* \)
with resulting contribution to FSG(\(H^*\)) [on top].

one would naturally convert from the action-graph to a transaction-
graph out of efficiency reasons.

Now refer to the definition/computation of the set \text{live-actions} in
Fig. A.2. Effectively it says that if a live \( R \)-action reads-something-
from a \( W \)-action, then that \( W \)-action and all its (eventual) earlier \( R \-
actions in the same transaction are live too. Considering that several
\( W \)-actions in a single transaction may be read-from by different live
\( R \)-actions, this implies that all these \( W \)-actions and all those \( R \-
actions in the same transaction occurring before any latest of these
\( W \)-actions, are actually live.

Still there may be some other \( W \)-actions occurring (before or) after
any of those above-mentioned latest live \( W \)-actions in the
Corresponding transaction. These will all be dead as they will neither
directly nor indirectly have any effects on the final database
results. Thus the \( R \)-actions occurring after any of those latest live
\( W \)-actions in the specific transaction, will also all be dead as their
values may definitely not be a basis for any lasting database results.
This is illustrated in Fig. A.5.

Fig. A.5. Illustration of \text{Live} and \text{Dead} \( R \)-actions in a transaction.
As examples we will use the following schedules:

\[H_1 = T_1: W_1(x)\]
\[T_2: R_2(z)W_2(z)R_2(x)W_2(y)\]
\[T_3: R_3(x)W_3(y)\]

\[H_2 = T_1: R_1(x)W_1(x)\]
\[T_2: R_1(y)R_1(z)W_1(z)\]
\[T_2: R_2(y)W_2(y)R_2(x)\]

Applying the FSG-construct, we may conclude that both schedule \(H_1\) and schedule \(H_2\) will be members of FSR. The results are:

\[H_1 \equiv_f T_2 \odot T_3 \odot T_1\]
\[H_2 \equiv_f T_2 \odot T_1\]

\(H_1\) is a not-read-before-write example. It this case we may have both dead \(W\)-actions and dead \(R\)-actions. \(H_1\) becomes a final-state serializable schedule because the update \(W_2(y)\) and the retrieval \(R_2(x)\) are dead actions. All the other retrievals and updates are live actions. (Note that schedule \(H_1\) is an extension of schedule \(H_{11}\) from Section 4.3).

\(H_2\) is a read-before-write example. It this case we may have dead \(R\)-actions only. \(H_2\) becomes a final-state serializable schedule because the retrieval \(R_2(x)\) is a dead action. All the other retrievals and updates are live actions.

The definitions used in [Ullm82] effectively work with live or dead transactions and not on the level of live or dead actions. In our context they actually say that if a (live or dead) \(R\)-action of a live transaction reads-something-from a \(W\)-action of another transaction, then that (entire) transaction is live too: i.e. that \(W\)-action and all earlier and later (or parallel) \(R\)-actions (and \(W\)-actions) are live. Or paraphrased; they say that just one live \(W\)-action (and then possibly one or more live \(R\)-actions) is enough to make the (entire) transaction live, and that all \(W\)-actions (and then necessarily all \(R\)-actions) have to be dead to make the transaction dead. This is not totally correct in view of the above discussions.

(Observe that in both the above examples we have to distinguish between live and dead actions and not only between live and dead transactions to assure membership in FSR).

But it is a non-fatal simplification. This means that the simplification does not lead to any non-serializable schedules being accepted as serializable, but only to some serializable schedules being considered as non-serializable. It stems from the fact that if some dead \(R\)-actions are treated as live, then more arcs (and arc-pairs) - and not fewer - is the result in the serialization graph-set FSG. This again leads to less chance - and not higher - of finding an acyclic serialization graph among FSG.
By the way, neither [Bern79b] nor [Papa79] discusses these points. This is only natural as the points are irrelevant in the single-action-multi-item model. Either the W-action and the R-action are both live or they are both dead. Further, neither [Date83] nor [Bern87b] mentions the points. This is again natural as these reference-materials do not cover the FSR-class. However, [Papa86] treats these points - though indirectly.
REFERENCES

[Adib80] M. Adiba, B.G. Lindsay:
"Database Snapshots"; In Proc. 6th Int. Conf. on Very Large Data Bases, 1980.

[Adib81] M. Adiba:
"Derived Relations: A Unified Mechanism for Views, Snapshots and Distributed Data"; In Proc. 7th Int. Conf. on Very Large Data Bases, 1981.

[A1sb76] P.A. Alsberg, J.D. Day:

[Atta84] R. Attar, P.A. Bernstein, N. Goodman:

[Bada79] D.Z. Badal:

[Bada81] D.Z. Badal:

[Banc85] F. Bancilhon, W. Kim, H.F. Korth:
"A Model of CAD Transactions"; In Proc. 11th Int. Conf. on Very Large Data Bases, 1985.

[Baye78] R. Bayer, R.M. Graham, G. Seegmüller (Eds.):

"Distributed Concurrency Control in Database Systems"; In Proc. 6th Int. Conf. on Very Large Data Bases, 1980.

[Baye80b] R. Bayer, H. Heller, A. Reiser:
[Beer86] C. Beeri, P.A. Bernstein, N. Goodman:


[Bern79a] P.A. Bernstein, N. Goodman:

[Bern79b] P.A. Bernstein, D.W. Shipman, W.S. Wong:

[Bern80a] P.A. Bernstein, N. Goodman:
"Timestamp-Based Algorithms for Concurrency Control in Distributed Data Base Systems"; In Proc. 6th Int. Conf. on Very Large Data Bases, 1980.

[Bern80b] P.A. Bernstein, D. Shipman:

[Bern80c] P.A. Bernstein, D.W. Shipman, J.B. Rothnie:
"Concurrency Control in a System for Distributed Databases (SDD-1)"; ACM Trans. on Database Systems 5(1), 1980.

[Bern81a] P.A. Bernstein, D.W. Chiu:

[Bern81b] P.A. Bernstein, N. Goodman:

[Bern81c] P.A. Bernstein, N. Goodman, E. Wong, C.L. Reeve, J.B. Rothnie:
"Query Processing in a System for Distributed Databases (SDD-1)"; ACM Trans. on Database Systems 6(4), 1981.

[Bern82] P.A. Bernstein, N. Goodman:
"A Sophisticate's Introduction to Distributed Database Concurrency Control"; In Proc. 8th Int. Conf. on Very Large Data Bases, 1982.
[Bern83a] P.A. Bernstein, N. Goodman:

[Bern83b] P.A. Bernstein, N. Goodman, V. Hadzilacos:
"Recovery Algorithms for Database Systems" In

[Bern83c] P.A. Bernstein, N. Goodman, M.Y. Lai:

[Bern84] P.A. Bernstein, N. Goodman:

[Bern86] P.A. Bernstein, N. Goodman:

[Bern87a] P.A. Bernstein, N. Goodman:

[Bern87b] P.A. Bernstein, V. Hadzilacos, N. Goodman:

[Bjor72] L.A. Bjork, C.T. Davies:
"The Semantics of the Preservation and Recovery of Integrity in a Data System"; TR TR-02.540, IBM Research Laboratory, 1972.

[Bjor73] L.A. Bjork:

[Blas79] M.W. Blasgen, J.N. Gray, M. Mitoma, T. Price:

[Bond76] J.A. Bondy, U.S.R. Murty:
"Graph Theory with Applications"; Macmillan, 1976.
[Borr81] A.J. Borr:
"Transaction Monitoring in Encompass: Reliable Distributed Transaction Processing"; In Proc. 7th Int. Conf. on Very Large Data Bases, 1981.

[Bray82] O.H. Bray:

[Buss82] U. Bussolati, G. Martella:

[Care84] M. Carey, M. Stonebraker:

[Casa81] M.A. Casanova:

[Ceri83a] S. Ceri, S.B. Navathe:

[Ceri83b] S. Ceri, G. Pelagatti:
"Correctness of Query Execution Strategies in Distributed Databases"; ACM Trans. on Database Systems 8(4), 1983.

[Ceri84] S. Ceri, G. Pelagatti:


[Chan82] A. Chan, S. Fox, W.T.K. Lin, A. Nori, D.R. Ries:

[Chan83] K.M. Chandy, L.M. Haas, J. Misra:
[Chen80] P.P. Chen, J. Akoka:

[Chu76] W.W. Chu:

[Chu82] W.W. Chu, P. Hurley:

[Coop82] E.C. Cooper:

[Dada80] P. Dadam, G. Schlageter:
"Recovery in Distributed Databases Based on Non-Synchronized Local Checkpoints" In S. Lavington (Ed.): "Information Processing 80", Proc. IFIP 4th World Computer Congress; North-Holland, 1980.

[Dani82] D. Daniels, P. Selinger, L. Haas, B.G. Lindsay, C. Mohan, A. Walker, P. Wilms:
"An Introduction to Distributed Query Compilation in R*"; In [Schn82].

[Date83] C.J. Date:
"An Introduction to Database Systems - Volume II"; Addison-Wesley, 1983.

[Date86] C.J. Date:

[Davi73] C.T. Davies:

[Davi85] S. B. Davidson, H. Garcia-Molina, D. Skeen:

[Daya78] U. Dayal, P.A. Bernstein:
"The Updatability of Relational Views"; In Proc. 4th Int. Conf. on Very Large Data Bases, 1978.
[Devo80] C. Devor, J. Weeldreyer:
"DCTS: A Tested for Distributed Database Research"; TR HR-80-268, Honeywell Corporate Computer Science Center, 1980.

[Dowd82] L.W. Dowdy, D.V. Foster:

[Draf80] I.W. Druffan, F. Poole (Eds.):
"Distributed Data Bases - An Advanced Course"; Cambridge University, 1980.

[Eage83] D.L. Eager, K.C. Sevcik:

[Effe84] W. Effelsberg, T. Haerder:

[E1Ab85] A. ElAbbadi, D. Skeen, F. Cristian:

[Elli77] C.A. Ellis:

[Epst78] R. Epstein, M.R. Stonebraker, E. Wong:
"Distributed Query Processing in a Relational Database System"; In Proc. ACM SIGMOD Int. Conf. on Management of Data, 1978.

[Epst80] R. Epstein, M. Stonebraker:
"Analysis of Distributed Data Base Processing Strategies"; In Proc. 6th Int. Conf. on Very Large Data Bases, 1980.

[Eswa76] K.P. Eswaran, J.N. Gray, R.A. Lorie, I.L. Traiger:

[Ferr82] A. Ferrier, C. Stangret:
"Heterogeneity in the Distributed Database Management System Sirius-Delta"; In Proc. 8th Int. Conf. on Very Large Data Bases, 1982.
[Garc82] H. Garcia-Molina, G. Wiederhold:
"Read-Only Transactions in a Distributed Database"; ACM Trans. on Database Systems 7(2), 1982.


[Garc88] H. Garcia-Molina, B. Kogan:

[Gare79] M.R. Garey, D.S. Johnson:

[Giff79] D.K. Gifford:

[Gill76] A. Gill:

"Granularity of Locks in a Shared Data Base"; In Proc. 1st Int. Conf. on Very Large Data Bases, 1975.


[Gray78] J.N. Gray:
"Notes on Database Operating Systems"; In [Baye78].

[Gray81a] J.N. Gray:
"The Transaction Concept: Virtues and Limitations"; In Proc. 7th Int. Conf. on Very Large Data Bases, 1981.

[Gray81b] J. Gray, P. Homan, R. Obermarck, H. Korth:
[Gray81c] J.N. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, I. Traiger:

[Gros80] J.M. Gross, P.E. Jackson, J. Joyce, F.A. McGuire:
"Distributed Database Design and Administration"; In [Draf80].

[Habe76] A.N. Habermann:

[Hadz86a] T. Hadzilacos, C.H. Papadimitriou:

[Hadz86b] T. Hadzilacos, M. Yannakakis:

[Hadz88] V. Hadzilacos:

[Haer83] T. Haerder, A. Reuter:

[Hami88] A.G. Hamilton:

[Hamm80] M. Hammer, D.W. Shipman:

[Holt72] R.C. Holt:

[Hopc79] J.E. Hopcroft, J.D. Ullman:
"Introduction to Automata Theory, Languages and Computation"; Addison-Wesley, 1979.
[Jens81] E.D. Jensen:
"Distributed Control"; In [Lamp81b].

[Kane85] P. Kanellakis, C.H. Papadimitriou:

[Kede81] Z.M. Kedem, A. Silberschatz:

[Kede83] Z.M. Kedem, A. Silberschatz:

[Kohl81] W. Kohler:

[Kort83] H.F. Korth:

[Kort86] H.F. Korth, A. Silberschatz:

[Kort88a] H.F. Korth, W. Kim, F. Bancilhon:

[Kort88b] H.F. Korth, G.D. Speegle:

[Kung80] H.T. Kung, P.L. Lehman:

[Kung81] H.T. Kung, J.T. Robinson:
"On Optimistic Methods for Concurrency Control"; ACM Trans. on Database Systems 6(2), 1981

[Kuss82] H. Kuss:
"On Totally Ordering Checkpoints in Distributed Databases"; In Proc. ACM SIGMOD Int. Conf. on Management of Data, 1982.

[Lamp76] L. Lamport:

[Lamp78] L. Lamport:

[Lamp81a] B.W. Lampson:
"Atomic Transactions"; In [Lamp81b].

[Lamp81b] B.W. Lampson, M. Paul, H.J. Siegert (Eds.):

[Lehm81] P.L. Lehman, S.B. Yao:

[LeLa81] G. LeLann:
"Motivations, Objectives and Characterization of Distributed Systems"; In [Lamp81b].

[Lin82] W.K. Lin, J. Nolte:

[Lind80a] B.G. Lindsay:
"Single and Multi-Site Recovery Facilities"; In [Draf80].

[Lind80b] B.G. Lindsay, P. Selinger:
"Site Autonomy Issues in R*: A Distributed Database Management System"; TR RJ2927 (36822), IBM Research Laboratory, 1980.

[Lind81] B.G. Lindsay:
"Object Naming and Catalog Management for a Distributed Database Manager"; In Proc. 2nd Int. Conf. on Distributed Computing Systems, 1981.
[Lind84] B.G. Lindsay, L.M. Haas, C. Mohan, P.F. Wilms, R.A. Yost:

"Sirius Systems for Distributed Data Management"; In [Schn82].

[Lome77] D.B. Lomet:

[Lori77] R.A. Lorie:
"Physical Integrity in a Large Segmented Database"; ACM Trans. on Database Systems 2(1), 1977.

[Lync83] N.A. Lynch:

[McLe81] G. McLean:
"Comments on SDD-1 Concurrency Control Mechanism"; ACM Trans. on Database Systems 6(2), 1981.

[Mena79] D.A. Menasce, R.R. Muntz:

[Mena80] D.A. Menasce, G.J. Popek, R.R. Muntz:

[Mino82] T. Minoura, G. Wiederhold:

[Moha83] C. Mohan, B. Lindsay:
[Morr85] R.J.T. Morris, W.S. Wong:

[Neuh82] E.J. Neuhold, B. Walter:
* "An Overview of the Architecture of the Distributed Data Base System Porel"; In [Schn82].

[Nygå86] M. Nygård et al.:

[Ober82] R. Obermarck:
"Distributed Deadlock Detection Algorithm"; ACM Trans. on Database Systems 7(2), 1982.

[Papa77] C.H. Papadimitriou, P.A. Bernstein, J.B. Rothnie:

[Papa79] C.H. Papadimitriou:

[Papa83] C.H. Papadimitriou:

[Papa84] C.H. Papadimitriou, P.C. Kanellakis:
"On Concurrency Control by Multiple Versions"; ACM Trans. on Database Systems 9(1), 1984.

[Papa85] C.H. Papadimitriou, M. Yannakakis:

[Papa86] C. Papadimitriou:

[Peas80] M. Pease, R. Shostak, L. Lamport:

[Pete85] J.L. Peterson, A. Silberschatz:

[Rand78] B. Randell, P.A. Lee, P.C. Treleaven:

[Reed79] D.P. Reed:

[Reut82] A. Reuter:

[Reut84] A. Reuter:

[Ries77] D.R. Ries, M. Stonebraker:

[Ries79] D.R. Ries, M. Stonebraker:

[Rose78] D.J. Rosenkrantz, R.E. Stearns, P.M. Lewis:

[Rose82] D.J. Rosenkrantz, R.E. Stearns, P.M. Lewis:

[Roth77a] J.B. Rothnie, N. Goodman:
J.B. Rothnie, N. Goodman:
"A Survey of Research and Development in Distributed Database Management"; In Proc. 3rd Int. Conf. on Very Large Data Bases, 1977.

J.B. Rothnie, P.A. Bernstein, S. Fox, N. Goodman, M. Hammer, T.A. Landers, C. Reeve, D.W. Shipman, E. Wong:
"Introduction to a System for Distributed Databases (SDD-1)"; ACM Trans. on Database Systems 5(1), 1980.

G. Schlageter:

H.J. Schneider (Ed.):
"Distributed Data Bases", Proc. 2nd Int. Symp. on Distributed Data Bases; North-Holland, 1982.

P.G. Selinger, M. Adiba:
"Access Path Selection in Distributed Data Base Management Systems" In
S.M. Deen, P. Hammersley (Eds.): "ICOD-1", Proc. 1st Int. Conf. on Data Bases; Heyden, 1980.

K.C. Sevcik:
"Comparison of Concurrency Control Methods Using Analytic Models" In

A. Silberschatz, Z. Kedem:

D. Sween:

D. Sween, M. Stonebraker:

D. Sween:
[Skee84] D. Skeen, D. Wright:

[Spac80] S. Spaccapietra:
"Heterogeneous Data Base Distribution"; In [Draf80].

[Spec83] A.Z. Spector, P.M. Schwarz:

[Stal85] W. Stallings:

[Stan77] D.F. Stanat, D.F. McAllister:

[Stea76] R.E. Stearns, P.M. Lewis, D.J. Rosenkrantz:

[Stea81] R.E. Stearns, D.J. Rosenkrantz:
"Distributed Database Concurrency Controls Using Before-Values"; In Proc. ACM SIGMOD Int. Conf. on Management of Data, 1981.

[Ston77] M. Stonebraker, E. Neuhold:

[Ston79] M. Stonebraker:

[Ston80a] M. Stonebraker:
"Homogeneous Distributed Data Base Systems"; In [Draf80].

[Ston80b] M. Stonebraker:
"Retrospection on a Database System"; ACM Trans. on Database Systems 5(2), 1980.
[Ston81] M. Stonebraker:

[Stro83] H.R. Strong, D. Doley:

[Tane81] A.S. Tanenbaum:

[Tay84] Y.C. Tay, N. Goodman, R. Suri:

[Tay85] Y.C. Tay, N. Goodman, R. Suri:

[Thom79] R.H. Thomas:

[Trai82] I.L. Traiger, J. Gray, C.A. Galtier, B.G. Lindsay:

[Ullm82] J.D. Ullman:

[Verh78] J.S.M. Verhofstad:

[Walt82] B. Walter:

[Wein82] P.J. Weinberger:
R. Williams, D. Daniels, L. Haas, G. Lapis, B. Lindsay, P. Ng, R. Obermarck, P. Selinger, A. Walker, P. Wilms, R. Yost:


E. Wong:


M. Yannakakis:

"The Various Notions of Correctness in Database Concurrency Control"; TR, Bell Laboratories, 1980.

M. Yannakakis:


M. Yannakakis:


M. Yannakakis:
