PROCEEDINGS / KONGRESOPPSOMMINGS

6th
SOUTHERN AFRICAN COMPUTER SYMPOSIUM

6de
SUIDELIKE-AFRIKAANSE REKENAARSIPOSIMUM

De Overberger Hotel, Caledon
2 - 3 JULY 1991

SPONSORED by
ISM
FRD
GENMIN

EDITED by
M H LINCK
Department of Computer Science
University of Cape Town
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foreword</td>
<td>1</td>
</tr>
<tr>
<td>Organising Committee</td>
<td>2</td>
</tr>
<tr>
<td>Referees</td>
<td>3</td>
</tr>
<tr>
<td>Program</td>
<td>5</td>
</tr>
<tr>
<td>Papers (In order of presentation)</td>
<td>9</td>
</tr>
<tr>
<td>&quot;A value can belong to many types&quot; B H Venter, University of Fort Hare</td>
<td>10</td>
</tr>
<tr>
<td>&quot;A Transputer Based Embedded Controller Development System&quot; M R Webster, R G Harley, D C Levy &amp; D R Woodward, University of Natal</td>
<td>16</td>
</tr>
<tr>
<td>&quot;Improving a Control and Sequencing Language&quot; G Smit &amp; C Fair, University of Cape Town</td>
<td>25</td>
</tr>
<tr>
<td>&quot;Design of an Object Orientated Framework for Optimistic Parallel Simulation on Shared-Memory Computers&quot; P Machanick, University of Witwatersrand</td>
<td>40</td>
</tr>
<tr>
<td>&quot;Using Statecharts to Design and Specify the GMA Direct-Manipulation User Interface&quot; L van Zijl &amp; D Mitton, University of Stellenbosch</td>
<td>51</td>
</tr>
<tr>
<td>&quot;Product Form Solutions for Multiserver Centres with Heirarchical Classes of Customers&quot; A Krzesinski, University of Stellenbosch and R Schassberger, Technische Universität Braunschweig</td>
<td>69</td>
</tr>
<tr>
<td>&quot;A Reusable Kernel for the Development of Control Software&quot; W Fouché and P de Villiers, University of Stellenbosch</td>
<td>83</td>
</tr>
<tr>
<td>&quot;An Implementation of Linda Tuple Space under the Helios Operating System&quot; P G Clayton, E P Wentworth, G C Wells and F de Heer-Menlah, Rhodes University</td>
<td>95</td>
</tr>
<tr>
<td>&quot;The Design and Analysis of Distributed Virtual Memory Consistency Protocols in an Object Orientated Operating System&quot; K Macgregor, University of Cape Town &amp; R Campbell University of Illinois at Urbana-Champaign</td>
<td>107</td>
</tr>
</tbody>
</table>
"Concurrency Control Mechanisms for Multidatabase Systems"
A Deacon, University of Stellenbosch 118

"Extending Local Recovery Techniques for Distributed Databases"
H L Victor & M H Rennhackkamp, University of Stellenbosch 135

"Analysing Routing Strategies in Sporadic Networks"
S Melville, University of Natal 148

"The Design of a Speech Synthesis System for Afrikaans"
M J Wagener, University of Port Elizabeth 167

"Expert Systems for Management Control: A Multiexpert Architecture"
V Ram, University of Natal 177

"Integrating Similarity-Based and Explanation-Based Learning"
G D Oosthuizen and C Avenant, University of Pretoria 187

"Efficient Evaluation of Regular Path Programs"
P Wood, University of Cape Town 201

"Object Orientation in Relational Databases"
M Rennhackkamp, University of Stellenbosch 211

"Building a secure database using self-protecting objects"
M Olivier and S H von Solms, Rand Afrikaans University 228

"Modelling the Algebra of Weakest Preconditions"
C Brink and I Rewitsky, University of Cape Town 242

"A Model Checker for Transition Systems"
P de Villiers, University of Stellenbosch 262

"A New Algorithm for Finding an Upper Bound of the Genus of a Graph"
D I Carson and O R Oellermann, University of Natal 276
FOREWORD

The 6th Computer Symposium, organised under the auspices of SAICS, carries on the tradition of providing an opportunity for the South African scientific computing community to present research material to their peers.

It was heartening that 31 papers were offered for consideration. As before all these papers were refereed. Thereafter a selection committee chose 21 for presentation at the Symposium.

Several new dimensions are present in the 1991 symposium:

* The Symposium has been arranged for the day immediately after the SACLA conference.
* It is being run over only 1 day in contrast to the 2-3 days of previous symposia.
* I believe that it is first time that a Symposium has been held outside of the Transvaal.
* Over 85 people will be attending. Nearly all will have attended both events.
* A Sponsorship package for both SACLA and the Research Symposium was obtained. (This led to reduced hotel costs compared to previous symposia)

A major expense is the production of the Proceedings of the Symposium. To ensure financial soundness authors have had to pay the page charge of R20 per page.

A thought for the future would be consideration of a poster session at the Symposium. This could provide an alternative approach to presenting ideas or work.

I would sincerely hope that the twinning of SACLA and the Research Symposium is considered successful enough for this combination survive. As to whether a Research Symposium should be run each year after SACLA, or only every second year, is a matter of need and taste.

A challenge for the future is to encourage an even greater number of MSc & PhD students to attend the Symposium. Unlike this year, I would recommend that they be accommodated at the same cost as everyone else. Only if it is financially necessary should the sponsored number of students be limited.

I would like to thank the other members of the organising committee and my colleagues at UCT for all the help that they have given me. A special word of thanks goes to Prof. Pieter Kritzinger who has provided me with invaluable help and ideas throughout the organisation of this 6th Research Symposium.

M H Linck
Symposium Chairman
SYMPOSIUM CHAIRMAN

M H Linck, University of Cape Town

ORGANISING COMMITTEE

D Kourie, Pretoria University.
P S Kritzinger, University of Cape Town.
M H Linck, University of Cape Town.

SPONSORS

ISM
GENMIN
FRD
# LIST OF REFEREES FOR 6th RESEARCH SYMPOSIUM

<table>
<thead>
<tr>
<th>NAME</th>
<th>INSTITUTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnard, E</td>
<td>Pretoria</td>
</tr>
<tr>
<td>Becker, Ronnie</td>
<td>UCT</td>
</tr>
<tr>
<td>Berman S</td>
<td>UCT</td>
</tr>
<tr>
<td>Bishop, Judy</td>
<td>Wits</td>
</tr>
<tr>
<td>Berman, Sonia</td>
<td>UCT</td>
</tr>
<tr>
<td>Brink, Chris</td>
<td>UCT</td>
</tr>
<tr>
<td>Bodde, Ryn</td>
<td>Networks Systems</td>
</tr>
<tr>
<td>Bornman, Chris</td>
<td>UNISA</td>
</tr>
<tr>
<td>Bruwer, Piet</td>
<td>UOFS</td>
</tr>
<tr>
<td>Cherenack, Paul</td>
<td>UCT</td>
</tr>
<tr>
<td>Cook Donald</td>
<td>UCT</td>
</tr>
<tr>
<td>de Jaeger, Gerhard</td>
<td>UCT</td>
</tr>
<tr>
<td>de Villiers, Pieter</td>
<td>Stellenbosch</td>
</tr>
<tr>
<td>Ehlers, Elize</td>
<td>RAU</td>
</tr>
<tr>
<td>Eloff, Jan</td>
<td>RAU</td>
</tr>
<tr>
<td>Finnie, Gavin</td>
<td>Natal</td>
</tr>
<tr>
<td>Gaynor, N</td>
<td>AECI</td>
</tr>
<tr>
<td>Hutchinson, Andrew</td>
<td>UCT</td>
</tr>
<tr>
<td>Jourdan, D</td>
<td>Pretoria</td>
</tr>
<tr>
<td>Kourie Derrick</td>
<td>Pretoria</td>
</tr>
<tr>
<td>Kritzinger, Pieter</td>
<td>UCT</td>
</tr>
<tr>
<td>Krzesinski, Tony</td>
<td>Stellenbosch</td>
</tr>
<tr>
<td>Laing, Doug</td>
<td>ISM</td>
</tr>
<tr>
<td>Labuschagne, Willem</td>
<td>UNISA</td>
</tr>
<tr>
<td>Levy, Dave</td>
<td>Natal</td>
</tr>
</tbody>
</table>
MacGregor, Ken
Machanick, Philip
Mattison Keith
Messerschmidt, Hans
Mutch, Laurie
Neishlos, N
Oosthuizen, Deon
Peters Joseph
Ram, V
Postma, Stef
Rennhackkamp, Martin
Shochot, John
Silverberg, Roger
Smit, Riel
Smith, Dereck
Terry, Pat
van den Heever, Roelf
van Zijl, Lynette
Venter, Herman
Victor, Hema
von Solms, Basie
Wagenaar, M
Wentworth, Peter
Wheeler, Graham
Wood, Peter

UCT
Wits
UCT
UOFS
Shell
Wits
Pretoria
Simon Fraser
Natal, Pmb.
Natal, Pmb
Stellenbòsch
Wits
Council for Mineral Technology
UCT
UCT
Rhodes
UP
Stellenbosch
Fort Hare
Stellenbosch
RAU
UPE
Rhodes
UCT
UCT
6TH RESEARCH SYMPOSIUM - 1991
FINAL PROGRAM

TUESDAY 2nd July 1991
10h00 - 13h00 Registration
13h00 - 13h50 PUB LUNCH

14h00 - 15h30 SESSION 1A
Venue: Hassner
Chairman: Prof Basie von Solms
14h00 - 14h30
"A value can belong to many types."
B H Venter, University of Fort Hare

14h30 - 15h00
"A Transputer Based Embedded Controller Development System"
M R Webster, R G Harley, D C Levy & D R Woodward, University of Natal

15h00 - 15h30
"Improving a Control and Sequencing Language"
G Smit and C Fair, University of Cape Town

15h30 - 16h00 TEA

SESSION 1B
Venue: Hassner C
Chairman: Prof Roelf v d Heever
14h00 - 14h30
"Design of an Object Orientated Framework for Optimistic Parallel Simulation on Shared-Memory Computers"
P Machanick, University of Witwatersrand

14h30 - 15h00
"Using Statecharts to Design and Specify the GMA Direct-Manipulation User Interface"
L van Zijl & D Mitton, University of Stellenbosch

15h00 - 15h30
"Product Form Solutions for Multiserver Centres with Hierarchical Classes of Customers"
A Krzesinski, University of Stellenbosch and R Schassberger, Technische Universität Braunschweig
16h00 - 17h30 SESSION 2A

Venue: Hassner

Chairman: Prof Derrick Kourie

16h00 - 16h30
"A Reusable Kernel for the Development of Control Software" W Fouché and P de Villiers, University of Stellenbosch

16h30 - 17h00
"An Implementation of Linda Tuple Space under the Helios Operating System" P G Clayton, E P Wentworth, G C Wells and F de-Heer-Menlah, Rhodes University

17h00 - 17h30
"The Design and Analysis of Distributed Virtual Memory Consistency Protocols in an Object Orientated Operating System" K MacGregor, University of Cape Town & R Campbell, University of Illinois at Urbana-Champaign

19h30 PRE-DINNER DRINKS

20h00 GALA CAPE DINNER
(Men: Jackets & ties)
WEDNESDAY 3rd July 1991

7h00 - 8h15 BREAKFAST

8h15 - 9h45 SESSION 3A

Venue: Hassner

Chairman: Assoc Prof P Wood

8h15 - 8h45
"Concurrency Control Mechanisms for Multidatabase Systems" A Deacon, University of Stellenbosch

8h45 - 9h15
"Extending Local Recovery Techniques for Distributed Databases" H L Victor & M H Rennhackkamp, University of Stellenbosch

9h15 - 9h45
"Analysing Routing Strategies in Sporadic Networks" S Melville, University of Natal

9h45 - 10h15 TEA

10h15 - 11h00 SESSION 4

Venue: Hassner

Chairman: Prof P S Kritzinger
Invited paper: E Coffman

11h00 - 11h10 BREAK
11h10 - 12h40 SESSION 5A
Venue: Hassner
Chairman: Prof C Bornman
11h10 - 11h40
"Efficient Evaluation of Regular Path Programs"
P Wood, University of Cape Town
11h40 - 12h10
"Object Orientation in Relational Databases"
M Rennhackkamp, University of Stellenbosch
12h10 - 12h40
"Building a secure database using self-protecting objects"
M Olivier and S H von Solms, Rand Afrikaans University

SESSION 5B
Venue: Hassner C
Chairman: Prof A Krzesinski
11h10 - 11h40
"Modelling the Algebra of Weakest Preconditions"
C Brink & I Rewitsky, University of Cape Town
11h40 - 12h10
"A Model Checker for Transition Systems"
P de Villiers, University of Stellenbosch
12h10 - 12h40
"A New Algorithm for Finding an Upper Bound of the Genus of a Graph"
D I Carson and O R Oellermann, University of Natal

12h45-12h55 GENERAL MEETING of RESEARCH SYMPOSIUM ATTENDEES
Venue: Hassner
Chairman: Dr M H Linck

13h00 - 14h00 LUNCH

FINIS 6th COMPUTER SYMPOSIUM
PAPERS

of the

6TH RESEARCH SYMPOSIUM
Concurrency Control Mechanisms for Multidatabase Systems

Andrew Deacon

Department of Computer Science
Stellenbosch University

Abstract

In conventional systems, transaction management has been thoroughly investigated and is now considered to be well understood. Transaction management in multidatabase systems, however, is less well understood. A multidatabase system (MDBS) is a type of distributed database system built from a collection of different centralized database systems (DBSs) located on sites in a computer network. The MDBS design goals differ from those of conventional DBSs in that a compromise is sought between the seemingly contradictory goals of creating a logically integrated database and allowing each component DBS to continue to function autonomously from the MDBS.

The conventional concurrency control correctness criterion for a DBS is serializability. It is, however, not always possible to guarantee serializability in a MDBS using the conventional approach. Several seemingly different solutions have been proposed. Using the nested transaction paradigm we provide a simple model which is used to develop a number of new MDBS schedulers by applying existing theory and concepts. The utility of our model is further illustrated by using it as a framework for explaining and justifying existing solutions, including those with weaker correctness criteria.

Key words and phrases: concurrency control, multidatabase systems, multi-level nested transactions, top layer schedulers, transaction processing model

1 Introduction

Database systems (DBSs) have effectively replaced file systems as the means of storing important and complex information used by an organization [Sib76]. The DBS simplifies the management of information by enabling data to be shared between users and providing control over different data copies. An important concept in DBSs is that of a transaction. A transaction is an execution of a program that accesses a shared database. The only way users are able to interact with the DBS is by creating and then submitting transactions. Transactions must appear to execute atomically. Ensuring the atomicity of transaction executions are the concurrency control and recovery problems. These problems have been extensively studied using a simple DBS model with considerable success (see [BHG87] for example).

With the proliferation of DBSs has come the need, particularly for large organizations, to access multiple databases. The drive for vendor flexibility and the suitability of different environments for specific application requirements results in these databases being stored on multiple heterogeneous and autonomous systems. A multidatabase system (MDBS) is a type of distributed DBS that is built up from a collection of heterogeneous centralized DBSs located on sites in a computer network. One of the MDBS design goals is to create a system that hides aspects of where the data are stored and how data are retrieved from or updated in any DBS. Thus the illusion is created of a logically integrated DBS. Each component DBS must be able to be inte-
grated into the MDBS without modifications and still be able to function autonomously.

Since an MDBS also supports the DBS transaction concept, similar types of concurrency control correctness criteria should ideally be adopted. Here we will discuss some of the problems in adopting these criteria and their implications. The presentation is not technical, our goal being to introduce a new transaction processing model for MDBSs and to show the utility of viewing an MDBS as a multi-level nested system.

First the traditional DBS transaction processing model is summarized. Next we highlight some problems when the simple DBS model is applied to MDBSs. The multi-leveled nested model is developed to provide solutions to the problems. Unfortunately implementations are difficult and expensive and in some applications the traditional correctness criterion may be too restrictive. In Section 4 some of the proposed weaker correctness criteria are discussed. Lastly problems relating to the integrating of component DBSs are discussed.

2 Concurrency Control in Conventional Database Systems

The correctness of any system must be defined relative to users’ expectations. Intuitively, a system is correct if it does what users want it to do. It is generally accepted that the correctness specification of concurrent executions in the context of DBSs is serializability [BH87]. From a user’s perspective, serializability essentially ensures that a sequence of atomic operations can be performed such that the overall effect is the same as would result if each user transaction was executed, in some order, one after the other [Pap79]. This notion of correctness is appealing, as the DBS need not know anything about the computations to be performed by a transaction.

First some important concepts in serializability theory are covered. Next we discuss the different scheduling mechanisms used in DBSs. Some variations, enhancements and extensions to the basic scheduling mechanisms are then discussed.

2.1 Serializability Theory

From the viewpoint of serializability theory, the only operations of a transaction that need to be modeled are those that access the database, namely the Read and Write operations. A database is viewed as a collection of named data items. Each Read and Write specifies the name of the data item read and written, respectively. In addition each completed transaction contains either a Commit or Abort as its last operation, indicating whether or not the transaction terminated successfully. We use \( r_i(x) \) to denote a Read operation, issued by transaction \( T_i \), on data item \( x \). Similarly \( w_i(y) \) denotes a Write operation, \( c_j \) the Commit of transaction \( T_j \), and \( a_i \) an Abort.

A *history* indicates the order in which operations, submitted on behalf of a transaction, were executed relative to one another. As some operations may be executed concurrently, a history is defined in terms of a partial order, as is a transaction. If two operations operate on the same data item and one of the operations is a Write, then they are said to *conflict*. A history must specify the order of all conflicting operations that appear in it. Two histories are equivalent if firstly they are defined over the same set of transactions and have the same operations, and secondly they order conflicting operations of committed transactions in the same way [BH87].

A complete history \( H \) is serial if, for every two transactions \( T_i \) and \( T_j \) that appear in the history, either all operations of \( T_i \) appear before all operations of \( T_j \) or vice versa. A history \( H \) is serializable (SR) if, considering only committed transactions, it is equivalent to a serial history \( H' \). Since serial executions are correct and because every SR execution, by definition, is equivalent to a serial one, every SR execution is also correct. The concurrency control problem is thus ensuring that all executions are SR [BG81].

Determining whether or not a history \( H \) over \( T = \{T_1, \ldots, T_n\} \) is SR is achieved by analyzing a graph derived from the history. This graph is called a serialization graph (SG). The SG for \( H \) is a directed graph in which nodes represent the committed transactions in \( H \). The edges are all \( T_i \rightarrow T_j \) \( (i \neq j) \) such that one of the operations
of \( T_i \) precedes and conflicts with one of the operations of \( T_j \). The Serializability Theorem, given in terms of these definitions, states that a history \( H \) is SR if and only if the SG of \( H \) is acyclic [BH87].

### 2.2 Concurrency Control Mechanisms

The question may be asked why transactions are not executed serially. Current computer systems have two basic types of storage: volatile and stable. Volatile storage is fast but expensive and data will be lost in the event of a power failure. Stable storage is cheap and usually has a large capacity but is very slow. While one transaction is accessing stable storage there will be an opportunity for the system to perform other operations. Greater multiprocessing is required to improve performance.

Concurrency control mechanisms must guarantee that all executions are SR while still being efficient. This is not achieved by adding synchronization mechanisms like semaphores, but by designing an algorithm, called a scheduler, that monitors the execution and intervenes to change the execution order of operations whenever necessary. The scheduler must perform its task at speeds comparable to the execution of the program [Pap83]. What is submitted to the scheduler is a sequence of accesses to the shared data.

#### 2.2.1 Two-Phase Locking

Locking is the most widely used concurrency control mechanism. This is because of its simplicity and generality. Each data item has a lock associated with it. Before a transaction \( T_i \) may access a data item, the scheduler first examines the associated lock. If no transaction holds a conflicting lock, then the scheduler obtains the lock on behalf of \( T_i \). If another transaction \( T_j \) holds a conflicting lock, then \( T_i \) has to wait until \( T_j \) gives up the lock. That is, the scheduler will not give \( T_i \) the lock until \( T_j \) releases it. The scheduler thereby ensures that only one transaction can hold a conflicting lock at a time, so only one transaction can alter the data item at a time.

To guarantee serializability all locks must be acquired before any are released [EGLT76]. Thus there are two phases, first acquiring locks and then releasing them. The locking mechanism is thus called Two-Phase Locking (2PL). If locks are all released when the transaction commits then the execution is said to be strict [BH87]. Strict executions simplify recovery by preventing cascading aborts.

An unfortunate property of 2PL schedulers is that they are subject to deadlocks. A deadlock is a circular waiting situation, involving two or more transactions, such that each transaction is waiting for another to complete. Deadlock can be described precisely in terms of a wait-for graph. A wait-for graph is a directed graph with transaction identifiers forming the nodes. An edge from transaction \( T_i \) to \( T_j \), denoted \( T_i \rightarrow T_j \), represents the fact that \( T_i \) waits for \( T_j \). The existence of a cycle in the wait-for graph indicates a deadlock situation. Maintaining and checking a wait-for graph is expensive; thus more efficient deadlock resolution mechanisms are used in practice [BH87].

#### 2.2.2 Timestamp Ordering

In timestamp ordering (TSO), the DBS assigns a unique timestamp to each transaction. Before another timestamp can be assigned, the counter is changed, thus ensuring that all timestamps are unique in the system. The DBS attaches a transaction's timestamp to each operation issued by the transaction. The TSO scheduler orders conflicting operations according to their timestamps. By enforcing this rule, we are ensuring that every pair of conflicting operations is executed in timestamp order. Thus, a TSO execution has the same effect as a serial execution in which the transactions appear in timestamp order. With TSO deadlock is avoided since no two transactions can be waiting for one another if the TSO rule is obeyed.

#### 2.2.3 Serialization Graph Testing

Serialization graph testing (SGT) schedulers maintain the SG of the history that represents
the execution it controls. The scheduler attains SR executions by ensuring the SG it maintains always remains acyclic. The SG maintained by the scheduler differs from the one described in Section 2.1 because many older transactions that cannot be part of cycles are not represented and active transactions are included to prevent cycles being formed.

### 2.2.4 Variations on the Basic Schedulers

Variations on the basic schedulers discussed above have been developed. This permits a scheduler to balance the costs between delaying executions and having to abort transactions whose executions are not SR.

An optimistic scheduler is based on the premise that it is sometimes easier to “apologize” than to “ask permission” [Her90]. In optimistic approaches, a transaction is allowed to execute without any operations being delayed, and then validated before completing to determine whether a conflict has occurred. In the case of an inconsistency, a transaction is aborted and then resubmitted. Any optimistic scheme is cost effective only if validation succeeds sufficiently often.

In a pessimistic scheduler, operations are synchronized before they are executed, to produce a SR execution. Pessimistic schedulers operate under the assumption that transaction conflicts will be frequent and if an inconsistency can occur, it will occur. These mechanisms avoid the overhead of aborting and redoing transactions which, if committed, would have resulted in a non-SR execution.

Pessimistic schedules can be further classed as aggressive or conservative. An aggressive scheduler tends to avoid delaying operations and will try to schedule operations as soon as possible. The opportunity to reorder operations may be lost, resulting in transactions having to abort. A conservative scheduler tends to delay operations, giving the opportunity to reorder operations received out of SR order, thus reducing the likelihood of causing a transaction to be aborted.

### 2.3 Multi-level Concurrency Control Mechanisms

A useful approach to design and description of complex DBSs is to decompose the system into a hierarchically organized collection of levels [BSW88]. This has been widely used in a number of systems where record level locking is crucial to obtain satisfactory throughput rates [ABC*76, PSS*87]. While page operations are the most efficient means of accessing stable storage, using page locks would result in an intolerable number of conflicts. By application of the multi-level paradigm, systems can be designed that process multiple non-conflicting record operations in parallel [BBG89].

In order to resolve page level conflicts, additional short term locks are needed. Since these locks are used only for isolating single record operations, they can be released after each record level action without violating serializability. Unfortunately recovery is made more complex since page-before images cannot be used to undo a partial execution. Consequently recovery actions must be performed by inverse operations on a level higher than the operation being undone [BSW88, PSS*87]. Cascading aborts are thus inevitable.

### 2.4 Distributed Schedulers

In a single processor system, the reliability of the system is closely related to the reliability of its processor. In a distributed system that consists of sites with their own processors, it might be possible to execute a transaction even after sites have failed by using the resources of other sites. 2PL can easily be used to build a distributed scheduler. In the remainder of this article we will assume Strict 2PL is used unless stated otherwise. Each site has a scheduler, based on 2PL, that manages data items stored locally. For reasons of simplicity we will assume that there is no data replication.

The schedulers at all sites, taken together, constitute a distributed scheduler. The task of the distributed scheduler is to process the operations submitted in a (globally) SR and recoverable
manner. In 2PL, a Read(x) or Write(x) is processed when the appropriate lock on x has been obtained, which only depends on what other locks on x are presently owned. Thus the scheduler at each site has all the information needed to decide when to process an operation, without requiring communication with other sites. However the release of locks is not as simple. To enforce the two-phase rule, a scheduler cannot release the locks of a transaction at one site until it knows that no other locks will be obtained at another site. If Strict 2PL is used, communication between sites can be avoided by integrating the locking release in the Commit.

In distributed DBSs, the Commit and Abort operations differ from those in centralized systems, in that these operations have to be collectively executed by all sites involved in an execution and still appear atomic to the user. In the absence of a general mechanism to manage replicated data, the only non-trivial problem is that of consistent termination [BHG87]. An algorithm that ensures consistent termination is called an atomic commit protocol (ACP). Guaranteeing the atomicity of a distributed transaction is to ensure that the entire transaction is either unanimously aborted or unanimously committed. A mixed decision can result in an inconsistent database [Ske81].

The most widely used ACP is the two-phase commit protocol. If all operations of T_i have executed successfully at a site, a precommitted state is entered, and a message is sent to the transaction's commit protocol coordinator. This forms the first phase of the protocol. Locks are then held until either a Commit or Abort operation is received. Only when the coordinator has determined that all sites where operations were executed are in a precommit state, is the Commit operation sent to each site. Otherwise Aborts are sent to all sites in a precommit state. This forms the second phase of the protocol. If, as a result of a failure a decision cannot be made to either commit or abort, locks must be held until a decision can be made.

3 Concurrency Control in Multidatabase Systems

Implementing an MDBS that executes transactions serially is straightforward. For example, all transactions may be executed one after the other or updates may be executed off-line. For many applications this is unsatisfactory. The solution is to permit concurrent executions. However guaranteeing that concurrent transaction executions are SR is difficult in an MDBS because MDBSs are not aware of all transactions executed on the shared data and implementing an ACP may not be possible [GL84]. Because of these problems the majority of MDBSs have limited support to distributed read-only transactions or updates at a single component DBS.

In the following subsections we investigate problems in developing a correct MDBS scheduler. First we consider implementing a distributed 2PL scheduler in an MDBS where component DBSs enable an ACP to be implemented. Concurrency control mechanisms other than locking could equally well have been used, but since locking is the most widely used mechanism in DBSs, we limit our discussion to a locking mechanism. Next a broader class of schedulers is developed where the component DBSs that do not support an ACP can be incorporated into the MDBS with the limitation that they be used only for retrieving data or that only one component DBS is updated.

3.1 Distributed Two-Phase Locking

Implementing the distributed 2PL mechanism discussed in Section 2.4 is only possible if the component DBSs have mechanisms that permit the construction of an ACP. On receiving a global transaction, the MDBS decomposes it into a number of subtransactions that can each be executed at a single component DBS. The results of these subtransactions are then integrated by the MDBS, by executing join, union, select and project operations, if we assume the relational model is used. Intermediate results may be sent to other component DBSs where additional subtransactions have to be executed. A final result
for the transaction is created and returned to
the user. All locks acquired during the execu-
tion must be held until commitment. Typically
each component DBS will support a Precommit
command that forms the first phase of the ACP.

As with all locking based mechanisms, dead-
lock is possible. However, the conventional solu-
tions to the deadlock problem cannot be used in
an MDBS as the following example illustrates.

Example 1
Consider an MDBS with two component DBSs
A and B. Data items x and y are at component
DBS A and z and u are at component DBS B.

Consider two global transactions: \(G_1\) which
updates \(x\) and \(u\); while \(G_2\) updates \(y\) and \(z\). In
addition let two local transactions execute: \(L_3\),
submitted to component DBS A, reads \(x\) and \(y\);
and \(L_4\), submitted to component DBS B, reads
\(z\) and \(u\).

We assume that the two global transactions
are each decomposed into two subtransactions
which will be denoted \(G_{1A}, G_{1B}, G_{2A}\), and \(G_{2B}\),
where the first element of the subscript denotes
the transaction identifier and the second the
component DBS where it is executed. To ensure
that the execution is SR and recoverable, each
subtransaction issues a Precommit after execut-
ing the last Write operation.

Consider the following execution where at A,
\(G_{1A}\) executes and is in a precommit state. Then
\(L_3\) obtains a read lock on \(y\) and waits for \(G_{1A}\)
to release its write lock on \(x\). Now \(G_{2A}\) is also
prevented from executing since it must wait for
\(L_3\) to complete before it can obtain the write lock
on \(y\). At B the reverse situation occurs, \(G_{2B}\) is
in a precommit state and \(L_4\) holds a read lock
for \(u\) which blocks \(G_{1B}\). Thus neither \(G_1\) nor
\(G_2\) is able to execute to completion which causes
deadlock.

In a conventional system it would have been
possible to construct the following wait–for
graph to detect the deadlock:

\[
\begin{array}{ccc}
L_4 & \rightarrow & G_2 \\
\downarrow & & \downarrow \\
G_1 & \leftarrow & L_3
\end{array}
\]

This deadlock situation cannot be detected by
the MDBS by constructing a global wait–for
graph because the MDBS is not aware of the lo-
cal transactions.

One solution is to modify the component DBS
to enable the MDBS to detect global deadlocks
[DL87]. This approach is undesirable because it
requires modifications to the component DBS,
which is generally not possible.

Various algorithms have been proposed that
prevent global transactions from entering states
in which global deadlock may occur. These ap-
proaches have relied on maintaining an acyclic
graph representing the component DBSs at
which subtransactions have been executed. Be-
fore another subtransaction can be issued a check
is performed to ensure that the new graph will
not become cyclic [RS82, BS88].

Timeouts provide a very simple deadlock res-
solution mechanism that has been widely im-
plemented, particularly in commercial systems
[BHG87]. Timeouts will cause at least one trans-
action to eventually abort. An advantage of us-
ing a timeout mechanism over a protocol that
prevents deadlock is that it does not rely on
the correct functioning of all components [RS82].
However, if the timeout period is not carefully
chosen and is too short, unnecessary transaction
aborts are possible. Unnecessary delays may re-
sult if the period is too long. A knowledgeable
person is typically required to select the timeout
period.

3.2 Top Layer Schedulers

The distributed 2PL scheduler discussed above
cannot be used in a network where data are
retrieved from component DBSs that do not
support a Precommit command, because oth-
wise the execution may not be equivalent to
a two–phase execution. We consider how non–
SR executions may be produced involving dis-
tributed read–only transactions. If these in-
consistent reads are used to update the compo-
nent databases then the MDBS may cause the
databases to become inconsistent.
3.2.1 Inconsistent Reads

Although all transaction executions at the component DBSs are SR, this does not guarantee that all global transactions are SR as the next example illustrates [BS88].

Example 2

Consider the same database described in Example 1. Assume that two global transactions: \( G_1 \) which reads \( x \) and \( u \); and \( G_2 \) which reads \( y \) and \( z \). There are no conflicting operations between the global transactions \( G_1 \) and \( G_2 \) according to the definition in Section 2.1.

In addition consider two local transactions: \( L_3 \), submitted to component DBS \( A \), reads and writes \( x \) and \( y \); and \( L_4 \), submitted to component DBS \( B \), reads and writes \( z \) and \( u \).

The histories in Figure 1 represent possible executions of the local transactions together with subtransactions, executed on behalf of global transactions, at the two component DBSs. In addition, the SG for the execution at each component DBS is given. The notation \( x_A \) is used to indicate that data item \( x \) is stored at \( A \).

If both local and global transactions were executed together in a conventional DBS, the following SG would have been created:

\[
SG(H) = G_1 \quad \xrightarrow{L_3} \quad G_2 \quad \xleftarrow{L_4}
\]

The presence of the cycle implies that the above execution is not SR. The two global transactions each reflect results that could be obtained if transactions were executed serially, but it would be impossible for the results to have been produced together. \( \square \)

Thus any two subtransactions executing together at the same component DBS can conflict, regardless of which data items were read. This was not unexpected since the executions of the global transactions are not two-phase. The 2PL policy requires that no locks may be released until all locks have been acquired. In Example 2, locks were released by each global transaction at one component DBS before each acquired additional locks at a second component DBS.

3.2.2 Multi-Level Model

The history, defined in Section 2.1, models transaction executions in conventional DBSs very well. However, it does not model MDBS transaction executions as successfully. The fact that each global transaction is executed as one or more subtransactions is not captured and thus cannot be exploited by a MDBS scheduler. To capture this information we define MDBS histories in a similar way to the nested history used in multi-level nested systems [BSW88, BBG89]. This view of MDBS transactions as being nested has been recognized by a number of authors, including Ries & Smith [RS82] and Gligor & Luekenbaugh [GL84]; however, they did not use this model to develop any new schedulers.

MDBS histories are represented by separate histories for each level. The global transactions represent top level transactions, with the second level being the local transactions and subtransactions submitted to the component DBSs. The bottom level represents the Read and Write operations executed by the component DBSs. An unusual characteristic is that local transactions do not appear on the top level.

In [BBG89] it is proven that a combined scheduler for an \( n \)-level system, consisting of \( n - 1 \) order preserving conflict based schedulers, used between the pairs of adjacent levels, is correct. This supports the observation made in Example 2 that it is insufficient to guarantee SR on only the bottom layer, where a layer is a pair of adjacent levels. A scheduler for the top layer is also required. The bottom layer scheduler is composed of the schedulers of the component DBSs and its operators are the Reads and Writes issued by the (sub)transactions submitted to the various component DBSs.
3.2.3 A Classification of Scheduling Mechanisms

The top layer scheduler must ensure that all the schedulers at the various component DBSs agree on a serialization order for every global transaction. Concurrency control mechanisms may be classified based on the serialization order they produce [BG81, Wei89]. Dynamic atomic algorithms, such as Strict 2PL, ensure that transactions are SR in the order in which they commit. Static atomic algorithms, such as TSO, ensure that transactions are SR in timestamp order. Some algorithms represent a combination of the dynamic and static algorithms [Wei89].

It is easier to build a top layer scheduler if all component DBSs use a static atomic algorithm since the serialization order can be anticipated by the MDBS from the order in which subtransactions are submitted to the DBS. Most component DBSs use 2PL, a dynamic atomic algorithm, which implies that the MDBS must wait until after subtransactions commit to be able to enforce a serialization order. If executions are not strict, the MDBS will have to wait for a period after commitment until it is certain that the component DBS could not serialize another transaction before it [BST89].

3.2.4 Subtransaction Conflicts

First we must define how subtransactions conflict in the top layer as has been done for Reads and Writes in the bottom layer. It is difficult for the MDBS to determine whether and how subtransactions conflict. Assuming executions are strict, if $G_i A$ was committed before $G_j A$ ($i \neq j$), then the MDBS can deduce that $G_i A$ preceded and could have conflicted with $G_j A$. To be sure that two subtransactions did not conflict they must be in a precommit state at the same time.

In order to guarantee that an execution is recoverable we assume that all subtransactions that issue Writes to a component DBS must participate in the ACP. Holding locks may result in deadlock. For reasons of simplicity we will assume that a timeout mechanism is used.

In Section 2 three different types of schedulers were mentioned. For each we can build an equivalent top layer scheduler. At the top layer, the operations are the subtransactions submitted to the component DBSs by global transactions.

3.2.5 Top Layer Serialization Graph Testing Scheduler

The Top Layer Serialization Graph Testing (TLSGT) scheduler maintains a type of SG which includes active transactions. This graph is
called the Stored Top Layer Serialization Graph (STLSG). An edge is added to an STLSG if any two operations (i.e., subtransactions) of different global transactions can conflict at a component DBS. Because we are not sure how subtransactions will be serialized before an execution, all possible orders must be considered. Thus for example if \( G_{iA} \) and \( G_{jA} \) are executed together then the two possibilities \( G_i \rightarrow G_j \) and \( G_i \rightarrow G_j \) must be considered. Thus several STLSGs will have to be maintained, all of which must be acyclic. The STLSGs for the execution in Example 2 are \( G_1 \rightarrow G_2 \), \( G_1 \leftarrow G_2 \) and \( G_1 \rightleftharpoons G_2 \). Clearly, in the worst case, there can be an exponential number of STLSGs.

When it becomes known how subtransactions were serialized, some edges in the STLSG may be deleted, which will enable other subtransactions to be scheduled. If, for all component DBS \( A \) where both \( G_i \) and \( G_j \) submit subtransactions, either \( G_{iA} \) commits before \( G_{jA} \) or both are in a precommit state, then \( G_i \) is assumed to precede and conflict with \( G_j \) and the edge \( G_i \rightarrow G_j \) can be deleted. Clearly, if at all component DBSs \( A \) \( G_{iA} \) and \( G_{jA} \) are in a precommit state, then both edges between \( G_i \) and \( G_j \) can be deleted. These deletions apply to all STLSGs and may result in some STLSGs becoming redundant. If, in Example 2, \( G_{1A} \) and \( G_{1B} \) had committed before \( G_{2A} \) and \( G_{2B} \) respectively then the edges in the STLSGs from \( G_2 \) to \( G_1 \) can be deleted. The only STLSG that remains is \( G_1 \rightarrow G_2 \).

Even after deleting edges between \( G_i \) and \( G_j \) the transactions may still be involved in a cycle of the form \( G_1 \rightarrow \ldots G_j \rightarrow \ldots G_i \), requiring that a transaction be aborted. If \( G_i \) has committed and has no incoming edges then this node and all associated edges may be deleted because it can never be involved in a cycle again.

The ADDS MDBS scheduler [BST87, BS88, BST89] is a variant of a pessimistic TLSGT scheduler with an aggressive scheduling policy [Dea91]. The ADDS scheduler is centralized. A single undirected graph, called a transaction graph is maintained, rather than possibly many STLSGs. In addition to guaranteeing serializability an acyclic transaction graph ensures that deadlock is avoided.

Since one may expect conflicts and deadlock to be rare a more optimistic approach would seem more appropriate. An optimistic scheduler permits subtransactions to execute that will create a cycle in the STLSG. This is done in the hope that the cycle forming edges will be deleted later. The cycles in the STLSG are only permitted while the transactions involved in the cycle can be aborted. The STLSG for all committed transactions may never have cycles. If the cycle forming edge cannot be deleted then a transaction must be aborted to break the cycle.

### 3.2.6 Top Layer Two-Phase Locking Scheduler

Using Top Layer Two-Phase Locking (TL2PL), the MDBS first requests a lock for a component DBS, before a subtransaction is executed. The locks are exclusive. The lock on component DBSs where subtransactions have precommitted or where commitment can be delayed, can be released once all subtransaction locks have been acquired. If subtransaction locks are released early then the component DBS lock must be held until the global transaction terminates. The locking policy must be two-phase, meaning that locks cannot be acquired after they have been released. This would appear to be contradicted by the release of the lock on a precommitted subtransaction. The locks are held, just the granularity has changed from a lock on all data items at a component DBS to those held by the subtransaction.

In the hope that the subtransactions will not conflict, an optimistic TL2PL scheduler can be developed that allows more than one subtransaction to be submitted to a component DBS, if the subtransaction can precommit or control when the locks will be released. This mechanism is the distributed 2PL if all component DBSs support a Precommit command. At component DBSs where the MDBS can only execute read-only subtransactions, because the DBS does not support a Precommit command, the lock on component DBSs must remain exclusive and cannot be released until the global transaction terminates.
3.2.7 Top Layer Timestamp Ordering Scheduler

A Top Layer Timestamp Ordering (TLTSO) scheduler executes subtransactions in timestamp order at each component DBS. The MDBS maintains a global clock. Global transactions submitted are given a unique ever-increasing timestamp which is attached to each subtransaction. If the TLTSO scheduler receives a subtransaction with a timestamp greater than that of the last subtransaction scheduled, the transaction must be aborted; otherwise the subtransaction can be submitted to the component DBS. By executing subtransactions in timestamp order deadlock is avoided.

A conservative TLTSO scheduler waits for an operation to be received from every site before submitting the operation to the component DBS. Thus no global transactions will be aborted due to a subtransaction arriving late. If a site has no subtransaction to send, a null subtransaction is sent.

An optimistic TLTSO scheduler permits multiple subtransactions to execute at a component DBS in the hope that they will precommit in timestamp order. If this does not happen the global transaction must be aborted.

4 Weaker Correctness Criteria

Enforcing serializability in an MDBS is expensive, restrictive and difficult without making assumptions about the component DBSs. For many MDBS applications it may be unnecessary to enforce serializability. A number of mechanisms are considered that are less restrictive in that they allow executions that would not be allowed by the schedulers discussed so far. These mechanisms sacrifice either some correctness or generality. In [KS88] three important mechanisms for enhancing concurrency that are not part of the traditional model are considered: multiple versions, nested transactions and explicit consistency predicates.

First we discuss how multiple versions may be exploited in an MDBS. Next three mechanisms, top layer state serializability, sagas and quasi serializability, are considered that view transactions as being multi-leveled. The quasi serializability mechanism also uses explicit consistency predicates.

4.1 Multiple Versions

Multiple transactions may be executed concurrently at the same component DBS but these transactions may delay other transaction executions. Various proposals for the more efficient processing of read-only transactions in distributed DBSs have been made [GW82, AS89]. Many of these algorithms have relied on multiple versions of data items being maintained, with the read-only transactions being able to read the older versions. This may be implemented in an MDBS if the component DBSs use multiple versions. Unfortunately multiple versions cannot easily be exploited in an MDBS to improve concurrency, since transactions cannot select which version of a data item they wish to read without possible sacrificing a large amount of stable storage.

4.2 Top Layer State Serializability

A state SR history satisfies the following [HP86]:

1. The history that includes the operations of all committed transactions leaves the database in a consistent state.

2. Each transaction sees a consistent database.

3. No Writes are lost.

A state SR history is not necessarily SR. For example, \( w_1[x] \ r_2[x] \ r_2[y] \ w_1[y] \) is state SR but not SR. A characteristic of state SR histories is if all operations of read-only transactions are deleted, then the history is SR. For the history above, deleting operations of the read-only transaction produces \( w_1[x] \ w_1[y] \), which is SR.

In many applications the type of anomaly in Example 2 is of little consequence. Permitting such executions requires no MDBS scheduling and thus can offer greatly improved performance. The majority of MDBSs currently adopt this approach by limiting support to read-only transactions. Only the component DBSs are able
to execute update transactions. The physical database state remains consistent, although the global transactions may reflect different views of its state. Using the multi-level model we can now recognize that the correctness criterion adopted in these MDBSs is top layer state serializability. Clearly deleting all read-only transactions from an MDBS history would make all MDBS histories trivially SR since there would be no global transactions left.

4.3 Sagas

The concept of a saga was proposed for situations where it is either impractical or not possible for a single transaction to execute atomically [GS87]. A saga is a transaction that is executed as a sequence of subtransactions that can be interleaved with other transactions. Either all the subtransactions that comprise the saga are executed successfully or compensating transactions are executed to remove the effects of the aborted saga. Of course other transactions may observe an inconsistent database state and the compensating transactions may not always be able to restore the database to a previously consistent state, but it is assumed that these cases are either rare or of little consequence. This fits into the multi-level nested model, with the saga transactions, the DBS transactions and DBS operations forming the three levels. The implementation of sagas does not require a top layer scheduler.

The saga concept can offer significant improvements in performance and it can be implemented using existing systems [GS87, DHL90]. It would appear that a saga mechanism can be implemented in an MDBS and will enable the component DBSs to function much more autonomously. An example of such an implementation is the semi-transaction concept in [EHHK88]. Here semi-transactions are used in developing an MDBS that integrates worldwide distributed autonomous databases for transnational accounting applications. This approach grew from a strong demand by the banks involved to preserve autonomy of their databases which is not possible if the traditional distributed DBS concepts are adopted.

A transaction can be considered a saga when a sequence of relatively independent steps can be identified where each step does not have to observe the same consistent database state. To amend partial executions of a saga $S_i$, an associated compensating transaction $C_i$ is required. The compensating transaction undoes, from a semantic point of view, the committed effects of $S_i$ but does not necessarily return the data items changed to the same state that existed before $S_i$ executed. This is analogous to the recovery mechanisms used in the multi-level systems discussed in Section 2.3. For example, to cancel a flight reservation a compensating transaction may decrement the number of reserved seats. It would in general be incorrect to return the number of booked seats to its original value since other reservations and cancellations may have been made. However, in the mean time another client may have been refused a seat because all seats were booked, while a subsequent client may have got a seat. This could be seen as unfair.

4.4 Quasi Serializability

Quasi serializability is a concurrency control correctness criterion for a top layer scheduling mechanism that is weaker that serializability [ED91]. As with our model, the definition of a quasi SR history is expressed in terms of global transactions only. However, this is done at the operations level rather than the subtransaction level.

A set of local histories is quasi serial if there exists a total order of global transactions such that for every $G_i$ and $G_j$, where $G_i$ precedes $G_j$ in the total ordering, $o_i$ precedes $o_j$ in a component DBS history for all $o_i$ of $G_i$, $o_j$ of $G_j$ and all component DBS histories [ED91]. The symbols $o_i$ and $o_j$ denote either a Read or Write operation issued by a subtransaction of $G_i$ and $G_j$ respectively. A history is quasi SR if it is equivalent to a quasi serial history.

Example 3

Reconsider the histories in Figure 1. The operations are executed in the order indicated by the arrows. Although the execution of lo-
5.1 Problems Integrating Component DBSs

Since both local and global transactions execute Reads on the shared data, schedulers that support high availability of replicated data and guarantee serializability, cannot be implemented in an MDBS. The primary copy, write-all-available and voting based approaches cannot easily be implemented correctly, without also requiring that all transactions that read the replicated data be executed by the MDBS, which violates the autonomy of component DBSs. The only approach that can be implemented correctly is the write-all approach [DL87, Dea91]. A problem with the write-all approach is that update transactions cannot execute immediately if all copies of the replicated data items being updated are not available [BHG87].

A second consequence of both local and global operations executing on the shared data is that global semantic integrity constraints cannot be enforced [Lar89]. This is because there can be no guarantee that the integrity constraints enforced by the MDBS and the component DBSs are equivalent.

In addition to the above, there are numerous additional problems in providing integrated functions including creating global schemas and enforcing security.

5.2 Seeking a Compromise

Because of the above problems, together with those of ensuring global deadlock freedom, serializability and recoverability, correctness criteria weaker than serializability are easily justified in an MDBS. The ideal MDBS should not attempt to provide the complete transparency and functionality, as for example in the list proposed by Stonebraker, for an ideal distributed DBS [Sto89]. Rather a compromise is needed between degrees of correctness, autonomy, data sharing and transparency that can be efficiently implemented and provides the type of functionality needed by users. One must be careful in seeking a compromise, that “correct” and “less correct” mechanisms are used together, since the...
Figure 2: The history and SG for a possible execution at $B$ in Example 2.

$H_B = r_4[z_2] \rightarrow r_4[u_B] \rightarrow w_4[z_B] \rightarrow w_4[u_B] \rightarrow c_4$

$SG(H_B) = G_2B \rightarrow L_4 \rightarrow G_1B$

5.3 Compromise Solutions

Using the multi-level model and the knowledge of which transactions are precommitted and those still executing at the same component DBSs we can build a Top Layer Wait-For Graph (TLWFG) to detect possible global deadlock situations (c.f. Section 2.2.1). The TLWFG for the execution in Example 1 is $G_1 \supseteq G_2$. A timeout mechanism is much less pessimistic than a top TLWFG testing mechanism or similar mechanisms mentioned in Section 3.1. Timeout mechanisms have the disadvantage that a knowledgeable person must select a suitable timeout period and transactions may be aborted although there is no deadlock. A compromise solution is to check the TLWFG after a timeout period has expired to determine if global deadlock is possible [BST90]. A decision may then be made by the MDBS to either abort a global transaction or reset the timeout period. This can make choosing a suitable timeout period less difficult.

Various MDBSs proposed the use of an “optimistic commit protocol” that uses a top layer scheduler mechanism to ensure that maximally one subtransaction can update a component DBS’s database [BST87, TTC*90]. This can reduce the risk of the database becoming corrupted. It operates under the assumption that all subtransactions can commit; if one aborts, it must be re-executed until it commits. The component DBSs thus need not support a Precommit command. This mechanism has the disadvantage that it cannot tolerate the failure of component DBSs, since a failure may make it impossible for the MDBS to either determine or control which transactions are aborted or committed by a component DBS. It seems inappropriate to use such a “less correct” mechanism together with one that enforces SR read transactions as was proposed for ADDS [BST87].

One restriction that will enable the enforcement of global integrity constraints is to restrict the component DBSs to updating certain data items; while other data items can only be updated by the MDBS [ED91]. Integrity constraints can then be enforced on these disjoint sets of data items, since the problem of ensuring the constraints enforced by the component DBSs and the MDBS are equivalent, is now eliminated. If multiple global schemas can exist, supporting global integrity may be inappropriate [DL87]. It is difficult to support a single global schema if a flexible mechanism for updating replicated data is not available, since updating it then becomes problematic.

Even if the correctness criteria are weakened, it is very difficult to implement schedulers that offer high availability for replicated data. The primary copy approach represents a compromise which offers users the choice between reading the primary copy and a possibly out-of-date non-primary copy [DAOT85]. Making other restrictions on distributed reads, a more flexible primary copy mechanism can be implemented [GA87].
6 NRDStar MDBS Project

NRDStar is an MDBS currently being developed in the Department of Computer Science at Stellenbosch University. At present support is limited to read-only transactions. The results of the research reported here are being applied to extend support to updates. It is our intention to implement several mechanisms, including distributed 2PL, saga and primary copy mechanisms. When NRDStar is loaded into a new environment, one or more of these mechanisms may be installed, depending on the user’s application. In addition, NRDStar may either be installed with either only one or multiple global schemas. This will allow NRDStar to be used for many applications which have widely varying correctness criteria.

7 Conclusion

The multi-level nested transaction model was successfully used to model MDBS transaction executions. The model enabled a number of new MDBS schedulers, called top layer schedulers, to be developed. This was achieved using existing theory and concepts; we only had to define how subtransactions conflict. These schedulers can be proven correct using the theory in [BBG89]. Unfortunately, with the exception of distributed 2PL, none of these scheduling mechanisms appears to be very practical. This supports the comment in [SL90] that it is unlikely that there exists a theoretically elegant solution that guarantees serializability without sacrificing performance and availability.

Several weaker correctness criteria and restrictions on processing have been proposed. The majority of MDBSs currently limit support to read-only transactions and perform no concurrency control. Applying the multi-level model we were able to recognize the correctness criterion used in these systems as being top layer state serializability. Update transactions are supported in the saga mechanism and mechanisms that enforce quasi serializability. A saga is a nested transaction executing in a system without a top layer scheduling mechanism. Limited recoverability is achieved using compensating transactions. Quasi serializability is a weak top layer correctness criterion that is correct as long as there are no value dependencies across component DBSs.

In a MDBS, support for recovery, global deadlock resolution, replicated data management, global semantic integrity control etc. may also be needed. However, as with concurrency control it does no appear that efficient general solutions are likely to be found. More restrictive compromise solutions have to be sought that are compatible with one another, do not violate the autonomy of component DBSs, can be implemented efficiently and that will solve users’ problems. The solution space has not been fully explored, which will require continued research [SYE*90]. We feel our model can serve as a useful basis for such research.

8 Acknowledgements

We would like to thank Martin Rennhackkamp and the anonymous referees for their constructive comments which led to an improvement in the clarity of this paper.

References


A. P. Sheth and J. A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous


