1997

Hybrid concurrency control for prioritized rules in active database systems

Chae-Kyu Kim

University of Wollongong

Recommended Citation
NOTE

This online version of the thesis may have different page formatting and pagination from the paper copy held in the University of Wollongong Library.

UNIVERSITY OF WOLLONGONG

COPYRIGHT WARNING

You may print or download ONE copy of this document for the purpose of your own research or study. The University does not authorise you to copy, communicate or otherwise make available electronically to any other person any copyright material contained on this site. You are reminded of the following:

Copyright owners are entitled to take legal action against persons who infringe their copyright. A reproduction of material that is protected by copyright may be a copyright infringement. A court may impose penalties and award damages in relation to offences and infringements relating to copyright material. Higher penalties may apply, and higher damages may be awarded, for offences and infringements involving the conversion of material into digital or electronic form.
HYBRID CONCURRENCE CONTROL FOR PRIORITIZED RULES IN ACTIVE DATABASE SYSTEMS

A thesis submitted in fulfillment of the requirements for the award of the degree

Doctor of Philosophy

from

UNIVERSITY OF WOLLONGONG

by

Chae-Kyu Kim

Department of Computer Science
March 1997
Declaration

This is to certify that the work reported in this thesis was done by the author, unless specified otherwise, and that no part of it has been submitted in a thesis to any other university or similar institution.

Chae-Kyu Kim
October 20, 1997
Abstract

In active database systems, rule processing occurs when an external transaction generates events. Certain rules are initially triggered by the events, and they are executed automatically when certain conditions are satisfied. Their execution can trigger other rules or the same rules, and so on-conceivably resulting in a finite set of triggered rules. The seemingly unpredictable activation of rules, however, produces two main problems: termination and confluence problems.

So far, most research efforts to solve these problems have concentrated on compile-time rule analysis. In compile-time rule analysis, it is difficult to determine which rules will be executed and what events will occur. It is therefore necessary to solve those problems at run-time. This thesis introduces a new transaction management protocol that allows not only deterministic execution but also terminated execution of rules in active database systems.

To solve the confluence problem, a strategy is introduced that eliminates the situation where different executions of rules in the finite set produce different results. This strategy is based on an assumption that there should only be one execution semantic for a given system of rules in the finite set. A unique execution semantic can be easily enforced by associating with each rule a priority that determines a global execution order. A property that execution of rules in the finite set is equivalent to execution of respective database transactions suggests a solution based on one of the transaction management techniques. The transaction management protocol enforces serializable executions of rules such that the respective serial execution order is consistent with an order determined by the rule priorities.

To solve the termination problem, this thesis uses a directed graph called a rooted
The rooted graph has a set of ordered pairs, called edges, representing the trigger relationship between two rules. An infinite cyclic execution is detected when the resulting rooted graph contains a cycle.

This thesis suggests a solution to the question: what should be done when a transaction attempts to access a data object in a mode that has the potential to violate a given serial execution order? The solution is to replicate both data and processes. Replication of data is a well known technique that requires a multiversion concurrency control mechanism. Replication of process is achieved by transaction cloning.

This thesis proposes two hybrid approaches to the construction of concurrency control algorithms for transactions in active database systems.

1. $H(n,k)$ concurrency control algorithms for external transactions in active database systems.

2. $H(l,m)$ concurrency control algorithms for respective transactions of rules in the finite set.

The parameter $n$ of $H(n,k)$ determines the acceptable size of cascading abort of external transactions and the parameter $k$ determines the total number of external transactions that execute concurrently with a particular external transaction. The parameter $l$ of $H(l,m)$ determines the maximum number of clone transactions of a rule transaction and the parameter $m$ determines the maximum probability that the database state on which a command is executed is changed in the future by other commands.

The two hybrid approaches have the ability to dynamically adjust the behaviour of concurrency control algorithms to changing environmental parameters. It is proved that for extreme values of parameters:

- as far as the concurrency control of external transactions is concerned, the $H(n,k)$ algorithm reduces itself either to strict two-phase locking algorithm or serialization graph testing.

- as far as the concurrency control of respective transactions of rules is concerned, the $H(l,m)$ algorithm reduces itself either to altruistic locking approach or to
serialization graph testing.
1. Chae-Kyu Kim and Janusz R. Getta, "Hybrid Concurrency Control in Database Systems", in Proceedings of Seventh Australasian Database Conference (ADC'96), (Editor: Rodney Topor) Volume 18, Number 2, pp 48-55, Melbourne, Australia, January 1996.


Acknowledgments

First, I would like to express my deepest gratitude to my supervisor Dr. Janusz R. Getta for his enthusiasm and support through the entire period of this study.

I am greatly indebted to my family (my mother, father, mother-in-law, father-in-law, brothers and sister-in-law) who have been a source of encouragement and support during the difficult times in this work.

I must thank my wife, Song-Chun Lee, and my daughters, Bo and Eon Kim, for their love and support. Their patience and understanding during the long hours of my work allowed me to finish this study.

I wish to express gratitude to Dr. Ki-Soo Sung, the Chancellor of Dong Myong University, Dr. Bong-Il Kim and Dr. Dan-Hyung Lee who encouraged my studying this area.

I would also like to thank Yi-Gun Yoo, Nam-Oh Kang, Bo-Sung Kim, Dr. Young-Jun Lee, Dr. Hun-Soo Jo and Yong-Ju Rho for their support.

Finally, I thankfully acknowledge the financial support provided by the Systems Engineering Research Institute (The President Gil-Lok Oh (Ph.D.)).
Contents

Abstract iv

Publications vii

Acknowledgments viii

I The Problem 1

1 Introduction 2
  1.1 Introduction ........................................... 2
  1.2 Problems ............................................. 4
  1.3 Previous works ....................................... 6
  1.4 Directions .......................................... 7
  1.5 Outline ............................................. 10

II Active Database Systems 11

2 Environment of Active Database Systems 12
  2.1 Introduction .......................................... 12
  2.2 Relational data model ................................ 12
  2.3 Conventional database management systems ....... 14
  2.4 Rule language ........................................ 16
  2.5 Architecture of active database system ............ 18
3 Transactions in Active Database Systems

3.1 Introduction .............................................. 21
3.2 Transaction boundary .................................... 22
3.3 Transaction execution model .............................. 25
3.4 Transaction execution ordering ........................... 27
3.5 Granularity .................................................. 28

4 Concurrency Control in Database Systems .......................... 30

4.1 Introduction .................................................. 30
4.2 Two main concurrency problems .............................. 31
4.3 Serializability ................................................ 34
4.4 Extended transaction model .................................. 36
4.5 Concurrency control algorithms .............................. 37
  4.5.1 Two-phase locking(2PL) .................................. 37
  4.5.2 Serialization graph testing(SGT) mechanism .......... 38
4.6 Hybrid concurrency control in database systems .................. 39
  4.6.1 Basic concept of hybrid approach ..................... 40
  4.6.2 Notations and definitions .............................. 41
  4.6.3 Scheduling algorithms .................................. 44
  4.6.4 $H(n,k)$ algorithms .................................... 46
  4.6.5 Conclusions .............................................. 49

III Hybrid Concurrency Control for Prioritized Rules in
Active Database Systems ........................................... 51

5 Introduction .................................................. 52

6 Basic Concepts ............................................... 57

  6.1 Commands vs atomic operations ............................ 57
  6.2 Rules ....................................................... 60
  6.3 Static analysis of rules .................................... 61
List of Tables
List of Figures

2.1 Rule execution in an active database system .................. 19

4.1 A sample serialization graph ........................................ 43
4.2 A sample extended serialization graph ............................. 43
4.3 Architecture of hybrid database system model .................... 44

5.1 Three cases when $z_j$ violates the order ......................... 55

6.1 An example of a rooted graph $G(e_i)$ .............................. 62
6.2 The execution order of commands of rule $r_i$ ...................... 63
6.3 The potential execution order of commands of rules in $G(e_i)$ 64

7.1 An example of directory ............................................. 67
7.2 Two clone transactions of transaction $z_j$ ....................... 73
7.3 The architecture of process replication ........................... 74

9.1 The architecture of an active database system model ............ 87
Part I

The Problem
Chapter 1

Introduction

1.1 Introduction

Active database systems have recently emerged as one of the most important directions in the evolution of the database technologies.

An active database system is a database system that provides a conventional database system with a user definable responsive functionality [DPG91, GJ92, Sto92]. Such functionality is typically defined by a set of rules that determine the actions to be executed when certain events happen and the respective conditions are satisfied [DBM88, DM89].

Thus, in comparison with conventional database systems that execute only operations explicitly submitted by the database applications, active database systems are capable of executing operations that cause the execution of other operations automatically.

The history of rule based systems can be traced back to ON construction defined in CODASYL [COD73] standard to express and enforce database integrity constraints. In a real-time database system [Rama92], rules are constructed by the form \((on \ E \ do \ A)\), where \(E\) is an event signalled by some executing program and \(A\) is a procedure that is invoked when \(E\) is detected. In artificial intelligence systems, rules have a form of implication \((if \ C \ then \ do \ A)\) [DRS92], where \(C\) specifies a predicate and \(A\) is a sequence of primitive operations.

In active database systems, the best known form of rule has a threefold pattern, known as ECA rule pattern [MD89]. Under this pattern a rule is consistent with the
1.1. Introduction

schema \( (\text{on } E, \text{if } C \text{ do } A) \), where

- the event part \( (E) \): specifies the event(s) that must happen for the rule to be executed.
- the condition part \( (C) \): determines a condition that should be satisfied to execute actions.
- the action part \( (A) \): contains a sequence of actions.

The ECA rule pattern was first introduced by the HiPAC project [DB88]. In the HiPAC, the semantics of ECA rule pattern states that whenever an event happens and the respective condition is satisfied then a sequence of actions is executed.

Active database systems which utilize ECA rules offer a general and powerful mechanism to solve a large variety of problems in the database management system context like the integrity constraint enforcement, maintenance of materialized views and derived data, replication management, and authorization checking, as well as serving as a basis for implementing large, efficient and flexible knowledge based systems, engineering information systems and expert systems.

They also offer a natural way to develop applications in which the shared properties of data are centralized instead of being scattered among application programs.

As an example, assume an inventory control system monitors an inventory database in order to place an order with a supplier for some part if its quantity falls below a threshold when it is updated. With conventional database systems which execute operations submitted by database application programs, every program that updates the quantity in stock of some part should check the condition and execute the ordering operation if necessary. With an active database system, the inventory database will be controlled automatically by the rules that execute ordering actions when the quantity of some part falls below a threshold. Thus the rules can be shared by many application programs.

It is recognized that in order to meet the growing demands placed on information processing systems, a rule interpretation system can be incorporated with conventional database systems [Cer92, CKTB95]. A large number of research projects are underway
to design and implement relational or object-oriented active database systems. HiPAC extended an objected-oriented database system with the paradigm of ECA rule pattern. Later on, the other active database systems such as Sentinel [CAMM94, CKTB95], Ode [GJ91], Adam [DPG91] and SAMOS [GD93], were developed for supporting active capability in the context of object-oriented databases. A number of research efforts have moved towards issues for supporting active capability in relational databases such as Starburst [PHH94], Ariel [Han92] and Postgres [SHP88]. The Starburst rule system [WF90] proposed by the IBM Almaden research center introduced set oriented ECA rules into an extensible relational database system. The Postgres rule system introduced tuple-oriented ECA rules into an extended relational database system.

1.2 Problems

The execution of a rule may create an event that may trigger another rule, whose execution may trigger the third one, and so on. This phenomenon is known as a cascading trigger. The ECA rule pattern offers only partial information about the entire rule process, since its event and its condition are explicitly specified, but the logic of its action part is hidden inside a user defined program. Since the logic of its action part is unknown, it is impossible to determine which operations will be executed and what events will be created. Therefore, the cascading trigger may cause the following two major problems during the rule execution:

- Confluence problem
  The execution of a set of rules triggered by the same event may provide different results each time it is executed depending on the rule execution order [Etz93, Etz94].

**Example 1.1** As an example, assume that two rules \( r \) and \( s \) are triggered and executed whenever data object \( z \) is updated.

<table>
<thead>
<tr>
<th>Rule ( r )</th>
<th>Rule ( s )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event: Update ( z )</td>
<td>Event: Update ( z )</td>
</tr>
</tbody>
</table>
1.2. Problems

Condition: True
Action: \( y := x + 3 \)

Condition: True
Action: \( y := x + 2 \)

As a result, the value of data object \( y \) is determined by the last rule that has been applied to calculate it.

- **Termination problem**

If one rule's action triggers another rule (or even itself again), and this rule's action triggers a third one, and so on, the uncontrolled execution of a set of rules may never terminate because of the cyclic execution of some rules. [WH95, BCP95].

**Example 1.2** As an example, assume that rule \( r \) is triggered and executed whenever data object \( x \) is modified and rule \( s \) is triggered and executed whenever data object \( y \) is modified.

<table>
<thead>
<tr>
<th>Rule ( r )</th>
<th>Rule ( s )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event: Update ( x )</td>
<td>Event: Update ( y )</td>
</tr>
<tr>
<td>Condition: True</td>
<td>Condition: True</td>
</tr>
<tr>
<td>Action: ( y := x + 3 )</td>
<td>Action: ( x := y + 2 )</td>
</tr>
</tbody>
</table>

As a result, both rules \( r \) and \( s \) trigger each other and generate a cyclic loop.

As shown in examples above, the confluence problem causes unexpected results generated by the non-deterministic execution order of rules and the termination problem causes an endless looping execution of a finite number of rules.
1.3 Previous works

The confluence problem has been the focus of much work recently [AWH92, WF90, KU94]. The [WF90] proposed a method which permits users to specify rule priorities. As a rule executes, events occur which trigger other rules. Triggered rules are placed in a pending rules set. Rules are selected from the pending rules set, and executed one at a time, according to their priorities. This causes a serial execution of rules.

In [AWH92], the concept of commutativity was used to determine whether rules triggered by the same event are confluent. For two rules, \( r \) and \( s \), triggered by the same event and having the same priority, the execution \( r \) and then \( s \), and vice versa, produces the same final database state if \( r \) and \( s \) are commutative. The commutativity of rules is determined by the static rule analysis process.

In general, a cyclic execution produces either an infinite number of possible database states or the infinite repetition of some database state. The former can be prevented if a rule designer gives some restrictions to the operation which causes the infinite cyclic execution. The latter can be detected if an operation which causes the infinite cyclic execution is executed on a database state repeated more than once.

Most research efforts to solve the termination problem have been concentrated on compile-time static analysis of rules [WH95]. Through the static rule analysis, a directed graph for rules is constructed, where nodes represent the rules and edges represent that the execution of a rule generates events which trigger other rules. A cyclic execution can be detected in the directed graph if one rule appears more than one time in a unique path connected by edges in the directed graph.

In [BCP95], a technique was proposed to detect the identical database state on which the same rule is triggered more than one time. The technique used run-time computations of checksums to represent a database state by means of a transition value. The run-time computations of checksums reduce the complexity to compare database states, but it is a burden for an active database system to produce the transition values of database states whenever a transition occurs.
1.4 Directions

The commutativity of rules determined by the static rule analysis process is not decidable. The decision as to whether two rules are commutative or not depends on the current database state on which those rules are executed. For two rules \( r \) and \( s \), assume that rule \( r \) inserts tuples into table \( t \) and rule \( s \) deletes tuples from the same table \( t \). By the static rule analysis process rule \( r \) and rule \( s \) are determined to be non-commutative since it assumes that \( r \)'s insertions can affect what \( s \) deletes. However, during the execution of \( r \) and then \( s \), if the tuples inserted by \( r \) never satisfy the delete condition of \( s \) then they actually do commute.

A technique which specifies the execution order of rules is efficient because the system can easily decide which rule to execute first. But this is only possible on a conceptual level because rules are executed serially according to the specified order given by a rule designer. At the implementation level, the serial executions may cause the system to make poor use of its resources, and cause inefficiency.

On the other hand, for a set of non-prioritized rules it may provide different results each time it is executed depending on the rule execution order. It is not possible to predict the execution order of those rules, since the system does not have any information about the rule execution order. Therefore, the final database state depends on the execution order of those rules and the order is arbitrary.

A strategy presented in this thesis not only permits rules to be executed depending on their priorities but also executes rules concurrently. The strategy is based on an assumption that the execution of rules is equivalent to the execution of the respective database transactions. This assumption leads to the notion of transaction management techniques.

In general, transaction management techniques enforce concurrent execution of transactions. When multiple transactions are executed concurrently, their operations may be interleaved. A history indicates the order in which the operations of the transactions were executed relative to each other. A serial history represents an execution in which there is no interleaving of the operations of different transactions.

One approach to the correctness of concurrent execution is to guarantee that any
interleaved execution of transactions produces the same effect as some serial execution of the same transactions. In this approach, a transaction management protocol is regarded as correct if all histories representing any interleaved executions that could be produced by it are equivalent to some serial history. Such histories are called serializable histories.

For two transactions $z_i$ and $z_j$, if there is no certain execution order for the two transactions, histories representing all possible executions of the two transactions are classified into two sets of serializable histories. Two kinds of serial histories are possible such that all operations of $z_i$ appear before all operations of $z_j$, and vice versa. This means that all serializable histories in the same set provide the same result. However serializable histories in different sets provide different results. Thus if a transaction management protocol enforces any serializable execution then whenever it executes the two transactions concurrently, it may produce different serializable histories and then the equivalent serial histories may be different.

From the active database systems point of view, for two rules, $r$ and $s$, triggered by either the same event or different events, if any execution order for these rules is not defined then the serial execution order for the respective transactions of these rules is arbitrary. Therefore, if rule $r$ and $s$ are executed within transaction $z_i$ and $z_j$, respectively the existence of the two kinds of serial executions causes the confluence problem.

In order to prevent this from occurring, assume there should be one execution semantics for a given system of a finite set of rules. An unique execution semantics can be easily enforced by a rule designer by associating with each rule a priority that determines a global execution order. More precisely, for two rules $r$ and $s$, if a given execution order is that $s$ should follow $r$ then all of the operations of $r$ should execute before any of the operations of $s$. If the transaction management protocol executes the respective transactions serially in the order determined by the priorities of rules, the same result can be produced whenever it executes the same transactions. Unfortunately, such serial executions are not efficient because they do not allow transactions to be interleaved at all.

Thus, it is necessary to look for a more efficient method than the serial execution
of transactions. The method should be based on the concurrent execution of transac-
tions. Any of the standard transaction management techniques can execute database 
transactions concurrently. However, standard transaction management techniques do 
not eliminate a situation where different executions of the same group of transactions 
provide different results, since they enforce any serializable executions of database 
transactions.

In active database systems rule processing occurs as a result of arbitrary database 
changes; certain rules are triggered initially, and their execution can trigger other rules 
or trigger the same rules, and so on-conceivably resulting in a finite set of triggered 
rules. A transaction management technique used in this thesis enforces serializable 
execution of database transactions such that the respective serial execution order is 
consistent with an order determined by the priorities of respective rules in the finite 
set. As a result, the confluence problem does not happen, since it produces the same 
results whenever it executes the same finite set of rule transactions concurrently.

The remaining problem is the question of how to detect a cyclic execution which 
produces the infinite repetition of some database state. A rule condition evaluation is 
either followed by the execution of its action or its action is not executed at all. The 
execution of its action can change the database state so that the condition of some 
other rule is no longer true. Thus, the problem of detecting the cyclic execution at 
compile-time is undecidable.

The transaction management system maintains a directed graph, called a triggering 
graph(TG) , at run-time. The nodes of TG are labelled with rule names. There is an 
edge, \( r \rightarrow s \), from node \( r \) to \( s \), if the action of rule \( r \) triggers rule \( s \). If a TG has a 
cycle: \( r \rightarrow s \rightarrow \cdots \rightarrow r \), then the initial database state at the beginning of the first 
rule \( r \) is compared with the database state at the beginning of the last rule \( r \). If these 
two database states are the same then the cyclic execution never terminates, otherwise 
the cyclic execution can be terminated at any stage.

To compare these two database states, the transaction management system uses a 
multiversion algorithm, since it does not destroy the old version of a database state 
when the database state is modified. This thesis proposes a method which examines 
whether or not the two database states are the same.
1.5 Outline

The rest of this thesis is composed of two parts: one part provides a brief overview of active database systems, the other part proposes a hybrid approach to the construction of a concurrency control algorithm for prioritized rules in active database systems. Part 2 consists of three chapters: Chapter 2 describes the active database system environment; Chapter 3 describes transactions which serve as a basis for defining when and how rules are executed; Chapter 4 provides a brief overview on concurrency control of transactions and introduces a hybrid approach to the construction of concurrency control algorithms for conventional database systems. Part 3 consists of the following chapters: Chapter 5 introduces the method of solving the open problems in active database systems; Chapter 6 deals with the basic concepts and introduces some notational conventions; Chapter 7 describes the execution environment of transactions in active database systems; Chapter 8 proposes a hybrid approach to the construction of concurrency control for prioritized rules in active database systems; Chapter 9 contains specifications of the scheduling algorithms that enforces the hybrid concurrency control strategy; Finally, Chapter 10 concludes the thesis.
Part II

Active Database Systems
Chapter 2

Environment of Active Database Systems

2.1 Introduction

This chapter discusses the active database system environment and defines an abstract model of an active database system. In the model an active database system incorporates a rule processing mechanism into a conventional database management system (DBMS). A conventional database management system is a collection of software modules that enables the user to create and maintain a database. One of its major components is a scheduler that controls the concurrent execution of database transactions so that the execution history is serializable. A rule processing mechanism is responsible for detection of the situations where the events happen and activation of the rules associated with the events. On the assumption that the execution of a rule is equivalent to the execution of the respective transaction, functions to be performed by the rule processing mechanism can be incorporated into a conventional database management system.

In this chapter, the concept of a relational data model is discussed first. Section 2.3 provides the internal structure of a conventional database management system. Section 2.4 discusses rule languages. Finally, an active database system model is proposed.

2.2 Relational data model

The main criterion used to classify conventional database management systems is the data model on which the conventional database management system is based. The data models used most often in current conventional database management systems are the
relational, network, hierarchical and object-oriented data models. In the data models, the description of a database and the database itself are separated. The description of a database is called the database schema. The data in a database at a particular moment in time is called a database instance.

This thesis uses the relational data model[Codd70] as a tool for describing the structure of a database. The relational data model has a simple and uniform data structure. It represents the data in a database as a collection of relations. When relations are thought of as tables, each table has a number of columns with unique names and each row in the table represents a collection of related data values. These values can be interpreted as a fact describing an entity. The table name and column names represent the meanings of the values in each row of the table. In relational database terminology, each row is called a *tuple*, each column name is called an *attribute* and a relation is called a table. The most important terms used in connection with data structure of the relational data model are defined as follows:

- **Table schema**
  A set of attributes of a table.

- **Primary key attribute**
  The value of a primary key attribute can be used to identify a tuple in a table. In general, more than one attribute may form the primary key of a table schema with the property that, at any given time, no two tuples of the table contain the same values in the primary key attributes.

- **Domain**
  Every attribute is defined on exactly one underlying domain, meaning that values of that attribute must be drawn from a domain, the domain is considered as a set of scalar values having the same type. All values of attributes are atomic, meaning that for every row and column position within a table, there always exists precisely one value, never a set of values.

- **Degree**
  The degree of a table is a total number of attributes in its schema.
2.3 Conventional database management systems

- Cardinality

The cardinality of a table is a total number of tuples in the table.

The following notations are used in this thesis.

**Notation 2.1** A table is denoted by the letter $t$ possibly followed by a subscript. The schema of table $t_i$ is denoted by $A(t_i)$ such that $A(t_i) = \{a_i, \ldots, a_n\}$.

**Notation 2.2** An attribute is denoted by the letter $a$ possibly followed by a subscript. A domain of attribute $a_i$ is is denoted by $d(a_i)$. An attribute $a_i$ of table $t_i$ is denoted by $t_i.a_i$.

**Notation 2.3** A state of table $t_i$ at a given time, denoted by $s(t_i)$, is defined as a subset of the Cartesian product of the domains of attributes in $A(t_i)$ and such that $s(t_i) \subseteq d(a_i) \times \cdots \times d(a_n)$, where $a_i, \ldots, a_n \in A(t_i)$.

**Notation 2.4** A database schema, denoted by $DB$, is a set of table schemas which describes the database such that $DB = A(t_1), \ldots, A(t_m)$. A database state, denoted by $s(DB)$, is a collection of all tuples of tables in $DB$ at a particular time.

2.3 Conventional database management systems

Conventional database management systems (DBMS) are passive in the sense that they execute only operations explicitly specified in the application programs. In conventional DBMS, operations are organized in transactions: collections of operations that are considered as atomic units for concurrency and recovery purposes [Pap86]. Conventional DBMS are responsible for executing operations concurrently, but the final result must be the same as a serial execution of the same operations of transactions.

An active database system is a database system which incorporates a rule interpretation mechanism into a conventional database management system. The rule interpretation mechanism is capable of monitoring changes in the state of a database. When an event happens, rules related to the event are selected for execution by the rule interpretation system. On the assumption that the execution of a group of rules is equivalent to the execution of respective database transactions, the conventional
database management system executes the respective transactions of rules which are selected by the rule interpretation mechanism.

This thesis uses an abstract model of the simplified internal structure of conventional DBMS. The term ‘abstract’ is used for the reason that the model does not have all functions which conventional DBMS perform.

In the model, a conventional DBMS consists of three modules: a transaction manager, a scheduler, and a data manager. Each module sends operations to, and receives replies from, the related module.

- **Transaction Manager(TM)**
  Transactions interact with the conventional database management system through the transaction manager. The transaction manager accepts the operations from the transactions and forwards them to the scheduler.

- **Scheduler**
  The scheduler controls the concurrent execution of operations submitted by the transaction manager. It performs one of the following three actions to help produce correct executions when it receives an operation from the transaction manager.

  - **Execute**: The scheduler sends the operation to the data manager for execution.
  
  - **Reject**: The scheduler refuses to process the operation and submits abort actions into its input queue for the transaction which issued the operation.
  
  - **Delay**: The scheduler puts the operation received from the TM in its delay queue.

- **Data Manager(DM)**
  The data manager receives an operation from the scheduler, and executes it over a database.
2.4 Rule language

Rules were originally introduced in the context of expert systems [BCP95] and in particular languages such as OPS5 rule language [BFKM85]. The OPS5 rule language constructs in the form \((\text{when } C \text{ then } A)\) where \(C\) is a predicate that is matched against data, and \(A\) is a sequence of actions that may update the data. This thesis considers Event-Condition-Action (ECA) rules. The data model into which ECA rules are incorporated is a relational data model. Commonly used data manipulation languages for relational database systems are extended to specify events, conditions and actions of ECA rules. As an example, the Postgress system [SHP88] provides a query language POSTQUEL for specifying data definition commands, queries and updates. The POSTQUEL extended its predecessor QUEL [Hel75]. The Ariel system [Han92] extended POSTQUEL with a rule language called Ariel Rule Language (ARL).

The general syntax for defining an ECA rule is:

\[
\text{define rule} \quad \text{rule-name} \\\n\text{on} \quad \text{event} \\\n\text{if} \quad \text{condition} \\\n\text{then} \quad \text{action}
\]

- The \textbf{define rule} clause of a rule defines a unique rule-name of the rule.

\textbf{Notation 2.5} \textit{In this thesis, a rule is denoted by the letter \(r\) possibly followed by a subscript which identifies this particular rule.}

- The \textbf{on} clause of a rule defines the types of events that may trigger the rule. It contains a primitive event or a composite event that must occur for the rule to be triggered. A primitive event may be defined as one of the database manipulation operations such as insert, delete, and update operations. To build up a composite event, primitive events can be combined by an operator such as disjunction, conjunction, and so on. The following primitive events can be specified after the \textbf{on} clause:


2.4. Rule language

- Database events
  The specification of database events corresponds to database manipulation operations such as inserting new tuples into a table, deleting existing tuples from a table, updating some attribute values of existing tuples, or selecting some attribute values of existing tuples.

- Temporal events
  Temporal events are of special interest in time-constrained applications and can be detected by a clock process.

- External events
  External events are detected by applications or components outside the database system.

The Starburst system[PHH94] supports only database events. The Postgres rule system supports database events and a few specific temporal events (e.g., time and date). The HiPAC system[Day88] proposed not only database events but also external events and temporal events. In this thesis and for simplification, only primitive events which are database events will be considered.

- The if clause of a rule defines conditions that are evaluated when the rule is triggered. In the NAOS system[CCS94], the if clause specifies predicates over the current database state. In this thesis, the condition is a collection of queries as in the HiPAC system and the OSCAR system[HFW90]. It is satisfied if all of these queries produce non-empty results.

- The action after the then clause of a rule is a sequence of operations to be performed when the rule is triggered and its condition is satisfied. In this thesis, the action can be a single data manipulation command or a sequence of data manipulation commands.

\textbf{Definition 2.1} The capital $C$ denotes a set of database commands and it is defined as follows:

$$C = \{(I, t_i, t) \mid t_i \in DB\} \cup \{(D, t_i, \psi) \mid t_i \in DB\}$$

$$\cup \{(U, t_i, a_k, \psi) \mid t_i \in DB\} \cup \{(R, t_i, \psi, t_j) \mid t_i \text{ and } t_j \in DB\},$$
where \( \psi \) is a tuple filter. It is a Boolean expression that determines the tuples to be deleted, to be updated or to be retrieved.

The Boolean expression is made up of a number of clauses of the form:

- \((t_i.a_i) \text{ (comparison op)} \text{ (constant value)} \text{ or} \)
- \((t_i.a_i) \text{ (comparison op)} (t_i.a_j)\)

where (comparison op) is normally one of the operators \{=,<,>,\geq,\leq,\neq\}.

The semantics of each command is as follows:

- \((I,t_i,t)\) is used to insert tuple \(t\) into \(t_i\).
- \((D,t_i,\psi)\) is used to delete tuples identified by \(\psi\) from \(t_i\).
- \((U,t_i.a_k,\psi)\) is used to update the value of attribute \(a_k\) of tuples identified by \(\psi\) from \(t_i\).
- \((R,t_i,\psi,t_j)\) is used to retrieve tuples identified by \(\psi\) from \(t_i\) and save them in temporary table \(t_j\).

**Notation 2.6** In the thesis, a command is denoted by the letter \(c\) possibly followed by a subscript.

As a result, the event part of ECA rule specifies when to check the rule, the condition part specifies what to check, and the action part contains what should be executed.

### 2.5 Architecture of active database system

This section provides an abstract model of the active database system. On the assumption that the execution of each rule is equivalent to the execution of respective database transactions, the active database system proposed in this section extends a relational database system. In Section 2.3, a model of the internal structure of a relational database management system was proposed. In the model, the relational DBMS consisted of three modules: a transaction manager, a scheduler, and a data manager.
In addition to these three modules, the active database system consists of a rule manager. The architecture of a sample active database system is illustrated in Figure 2.1.

As shown in Figure 2.1, the transaction manager receives operations issued by transactions and forwards them to the scheduler. Concurrency control for transactions is provided by the scheduler. After the scheduler receives an operation from the transaction manager, it enforces serializable execution of transactions such that the respective serial execution order is consistent with an order determined by rule priorities. The scheduler sends the operation to the data manager for execution. When the data manager finishes executing the operation, the data manager informs the scheduler and the scheduler relays it back to the transaction manager. When the transaction manager receives an acknowledgment of the execution of an operation from the scheduler, it produces an event message related to the operation and sends it to the rule manager. The rule manager is responsible for detecting rules associated with the event. In this DBMS, each rule detected by the rule manager is considered to be executing within an individual transaction. Thus, for each detected rule, the respective transaction of the
rule is created. By issuing a commit operation, the transaction terminates normally and all of its effects should be made permanent. By issuing an abort operation, the transaction terminates abnormally and all of its effects should be obliterated.
Chapter 3

Transactions in Active Database Systems

3.1 Introduction

This chapter describes transactions which serve as a basis for defining when and how rules are executed. In the model, transactions are expected to satisfy the following four conditions known as ACID properties [Gra81, HR83]:

- **Atomicity**
  Either all operations are executed or none of them.

- **Consistency**
  If a database is consistent before a transaction is executed, it is again consistent afterwards.

- **Isolation**
  It requires each transaction to observe a consistent database, i.e., not to read the intermediate results of other transactions.

- **Durability**
  After a transaction has finished, its effects are persistent.

As mentioned earlier, the rules considered in this thesis are ECA rules. The semantics of ECA rules states that whenever a rule is triggered, the rule's condition is evaluated and then the rule's action is executed if the rule's condition is satisfied. Section 3.2 describes how the event part, condition part and action part of a rule can be coupled to form transactions. It also describes when the condition part and the action part of a triggered rule are executed.
Section 3.3 discusses a brief overview of transaction execution models such as the flat transaction model and the nested transaction model.

As one event may trigger several rules, there is a need for ordering between the triggered ones. Section 3.4 introduces a policy to determine the execution order of rules.

Rules may be executed at several granularities such as a tuple or a set of tuples. Some active database systems support one granularity, others a mixture. Section 3.5 discusses how rules react to events caused by database modifications.

3.2 Transaction boundary

In active database systems, when a rule is triggered by an event generated by a transaction, three components of the rule need to be coupled to form transactions in some way. There are four different ways in which the event part, condition part and action part can be coupled to form transactions[CLJ91]:

- **ECA coupling**
  The three components of the rule triggered are coupled together in a single transaction.

- **EC-A coupling**
  The event part of the rule triggered is coupled into a single transaction with its condition part. But the action part is executed by a new transaction.

- **E-CA coupling**
  The condition part and action part of the rule triggered are coupled together in a single transaction. It is different from the transaction where the triggering event is generated.

- **E-C-A coupling**
  Each component of the rule triggered exists as a stand-alone transaction.

On the other hand, it is possible for a triggered rule that the rule’s condition can be evaluated in the same transaction as the triggering event or it can be evaluated in
a separate transaction. It is also possible for a triggered rule that the rule’s action can be executed in the same transaction as the rule’s condition evaluation or it can be executed in a separate transaction. To do this, HiPAC introduced *E-C and C-A couplings* [Day88]

- **E-C coupling**
  
  The *E-C coupling* is the relationship, relative to transaction boundaries, between the triggering event and the condition evaluation. There are three possible coupling modes:

  - An *immediate* mode
    
    When the triggering event occurs, the condition is evaluated immediately in the same transaction as the triggering event.
  
  - A *deferred* mode
    
    When the triggering event occurs, the condition is evaluated in the same transaction as the triggering event, but at the end of the triggering transaction before it commits.
  
  - A *separate* mode
    
    When the triggering event occurs, the condition is evaluated in a separate transaction.

- **C-A coupling**
  
  The *C-A coupling* specifies the relationship, relative to transaction boundaries, between the evaluation of the condition and the execution of the action. There are two possible coupling modes:

  - An *immediate* mode
    
    When the condition is evaluated to true, the action part is executed immediately in the same transaction as the condition part.
  
  - A *separate* mode
    
    When the condition is evaluated to true, the action part is executed in a separate transaction.
3.2. Transaction boundary

Sybase system [Syb90] supports the E-CA coupling. On the other hand Starburst system [WF90] and Postgres system [Sto92] support the E-C and C-A couplings. The Postgres, Sybase and ETM [KDM88] systems support only the immediate coupling mode whereas the Starburst active database system supports only deferred coupling mode.

In the E-CA coupling, two situations are possible [CJL91]: one is that the condition is evaluated and the action is executed in the same transaction where the triggering event is generated, the other is that the condition is evaluated and the action is executed in a single transactions which is different from the transaction where the triggering event is generated. In this thesis, for a triggered rule, the condition part and action part of the rule are coupled together in a single task, and the following three coupling modes specify the relationship, relative to transaction boundaries, between the transaction where the task is executed and the transaction where the triggering event is generated.

- An immediate mode
  When the triggering event occurs, the condition is evaluated immediately in the same transaction as the triggering event. If the condition is satisfied then the action is executed immediately in the same transaction where the condition is evaluated.

- A deferred mode
  The condition is evaluated in the same transaction as the triggering event, but after the final operation of the triggering transaction terminates. If the condition is satisfied then the action part is executed immediately in the same transaction as the triggering event.

- A separate mode
  The condition is evaluated in a separate transaction as soon as the operation that produced the triggering event terminates. If the condition is satisfied then the action part is executed immediately in the same transaction where the condition is evaluated.

Notation 3.1 A transaction is denoted by the letter z possibly followed by a subscript.
3.3 Transaction execution model

An event can trigger several rules. Therefore it is necessary to support a rule execution model that supports concurrent as well as prioritized serial execution of rules. On the other hand, the introduction of coupling modes requires an appropriate transaction execution model.

This section provides a brief overview of two transaction execution models: one is a flat transaction execution model, the other is a nested transaction execution model.

- The flat transaction execution model

In the flat transaction model, there are two types of tasks: one is external tasks, the other is internal tasks [CJL91].

- External tasks are tasks that contain database operations that arrive at the active database system from an application program.

- Internal tasks are tasks resulting from the triggering of rules. Once a rule is triggered, a condition task evaluates the rule's condition. If the condition evaluates to true, an action task is generated to perform the rule's action.

The execution of an external task may trigger several rules and the execution of their action tasks may trigger other rules, and so on. Depending on the coupling modes of rules, the internal tasks of these rules are either directly or indirectly related to the external task. Thus, a single external task could cause an arbitrary long chain of internal tasks.

In the flat transaction execution model, these internal tasks are executed in the same transaction as the external task [Car91, LS95]. As a result, the external task and all the internal tasks are coupled together in a single transaction.

- The nested transaction execution model

In the nested transaction model, there are two types of transactions: one is nested transactions (also called sub-transactions); the other is top-level transactions [Day88].
3.3. Transaction execution model

- Sub-transactions
  When a rule is triggered by an event, a transaction is created and the rule is executed in that transaction. The rule’s coupling mode specifies the parent\child relationship between the two transactions. If the rule’s coupling mode is immediate or deferred, then the transaction is a sub-transaction of the transaction which generated the triggering event. The sub-transaction is called a child transaction of the transaction containing the triggering event.

- Top-level transactions
  Top-level transactions are transactions which have no sub-transaction. When a rule is triggered by an event, if the rule’s coupling mode is separate, then the rule is executed in a top-level transaction.

As a result, in the nested transaction model, a user-defined transaction or a triggered rule where the coupling mode is separate are executed in a top-level transaction. Otherwise, rules triggered by an event are executed in the nested transactions of the triggering transaction.

Flat transaction model is used in some active database systems such as the Starburst system [AWH92] and the A-RDL system [SK95]. In this model, if the execution of any internal task in a transaction fails then the entire transaction aborts.

Nested transaction model is used in some active database systems such as HiPAC and Sentinel[CAMM94]. In the nested transaction model, if a transaction aborts then its sub-transactions also abort.

In this thesis, rules triggered by an event may be executed in individual transactions. The respective transaction of a rule contains operations in the action-part of the rule and conditions in the condition-part of the rule. The execution of operations in the action-part of a rule may generate new events that may trigger other rules and so on. This cascading trigger phenomenon can be represented by a hierarchical structure. In the hierarchical structure, an external transaction appears as a top-level transaction and the respective transaction of triggered rules appears one level farther away from the top-level transaction than the triggering transaction. To manage the execution order of transactions, this thesis uses a potential execution order of transactions in the
hierarchical structure. The potential execution order of a transaction is given by a rule designer. This thesis assumes that operations of transactions which lie on a unique path between two transactions are treated as a global transaction.

3.4 Transaction execution ordering

On the assumption that the execution order of rules in active database systems is the same as the execution order of respective database transactions, this section provides a brief overview of rule execution order.

As events can trigger several rules or as triggered rules may not be executed immediately after the occurrence of their triggering event, it is necessary to determine the order in which they are to be executed.

This section discusses two methods: one is to support serial execution of rules, the other is to support concurrent execution of rules.

- Sequential execution
  In the serial execution method, rules triggered by the same event are assigned to priorities based on either a partial order or a total order. Users are able to prioritize rules, so the intended execution order will be obeyed during execution.
  - A partial execution order (e.g., defining the happens-before relationship between rules).
  - A total execution order (e.g., indicating rule priorities as integer numbers, such that each rule in the system has a unique natural number as its priority).

- Concurrent execution
  In the concurrent execution method, users do not determine the order in which two rules triggered by an event are to be executed. When no ordering is specified between two rules, their execution order is arbitrary. The concurrent execution method enforces serializable execution of a set of rules such that it has the same effect as a possible serial execution of the same set of rules.
As an example, for two rules \( r_i \) and \( r_j \), triggered by an event, if their execution order is arbitrary, there exist two possible serial executions: one is \( r_i \) executes before \( r_j \), the other is \( r_j \) executes before \( r_i \). Serializable executions of two rules are categorized into two groups: one has the same effect as the former, the other has the same effect as the latter.

As a result, if multiple rules are triggered at the same time during rule processing, and their execution order is arbitrary, then the final database state at termination of rule processing depends on which serializable execution is considered (i.e., the confluence problem).

This thesis chooses the concurrent execution model but the final database state depends on a given serial execution order determined by a rule designer.

### 3.5 Granularity

Rules may be executed at several granularities such as a tuple or a set of tuples. If the granularity is a tuple level, rules are triggered separately by each distinct updated tuple. If the granularity is a set level, rules are triggered only once by the collective modification of tuples. In this thesis, rules are triggered by a primitive event and the primitive is generated after the corresponding operation is executed at the set level.

**Example 3.1** As the following example illustrates, the semantics can be quite different for the different granularities.

\[
\begin{align*}
\text{Rule } r & \quad \text{Rule } s \\
\text{Event: Delete EMP} & \quad \text{Event: Insert EMP} \\
\text{Condition: } \text{Emp.sal} \geq 100,000 & \quad \text{Condition: True} \\
\text{Action: Emp.sal := Emp.sal } \times 0.9 & \quad \text{Action: Delete from EMP} \\
& \quad \text{where EMP.sal } \geq 100,000 \\
\end{align*}
\]

\[
\begin{align*}
\text{EMP table} & \quad \text{EMP-No} & \quad \text{Sal} \\
E1 & \quad \text{100,000} \\
\end{align*}
\]
Assume that rule $s$ was triggered by an event. If the condition of rule $r$ is evaluated at the set level granularity, (i.e., after rule $s$ deletes the entire tuples in EMP table), then the condition of $r$ evaluates to false. If the condition of rule $r$ is evaluated at each tuple deleted by $s$, then the condition of $r$ evaluates to true.
Chapter 4

Concurrency Control in Database Systems

4.1 Introduction

This chapter provides a brief overview on concurrency control of transactions. As mentioned earlier, a database is a collection of related data. Users interact with the database through application programs that perform the functions desired by the users. The execution of such programs results in a partially ordered set of operations. This set of operations is called a transaction.

Transactions are expected to satisfy the ACID properties [HR83]: The ACID properties of transactions are usually ensured using two main means: one is known as a concurrency control, the other is known as a recovery. The concurrency control is the activity aimed at preserving the consistency property when transactions are executed concurrently. Mechanisms that ensure the consistency property of transactions are called concurrency control mechanisms. The recovery is the activity for the purpose of preserving the property that a transaction either executes in entirety or not at all even if there are partial system failures while executing the transaction. Mechanisms that ensure the all-or-nothing as well as isolation and durability properties are called recovery mechanisms.

Section 4.2 discusses two main concurrency problems that arise when a set of correct transactions are allowed to execute concurrently. Section 4.3 describes the concept of serializability used to prove the correctness of concurrency control. In section 4.4, an extended transaction model is discussed and concurrency control algorithms are described in section 4.5.
4.2 Two main concurrency problems

Given a correct database state, an individually correct transaction will produce another correct database state as output if it is executed in isolation. In a multi-user system, however, when two or more individually correct transactions execute concurrently, their operations execute in an interleaved fashion. That is, operations from one individually correct transaction may execute in between two operations from another individually correct transaction. This interleaving can cause transactions to behave incorrectly, thereby leading to an inconsistent database state. The following two problems are the best known concurrency problems [BHG87]:

- **The lost update problem**
  
  This problem occurs whenever two transactions attempt to update an attribute value in a tuple that is retrieved by the two transactions before either of them update the attribute’s value.

**Example 4.1** Suppose the following two transactions such that transaction zi decreases the salary of E1 by 10 percent, and transaction zj decreases the salary of E1 by 20 percent.

**Transaction zi.**

Read from Emp such that Emp.no = E1

Update Emp.sal := Emp.sal × 0.9

**Transaction zj.**

Read from Emp such that Emp.no = E1

Update Emp.sal := Emp.sal × 0.8

<table>
<thead>
<tr>
<th>EMP table EMP.no EMP.sal</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
</tr>
<tr>
<td>100,000</td>
</tr>
</tbody>
</table>
4.2. Two main concurrency problems

Consider as an example of the lost update the following interleaved executions of the two transactions $z_i$ and $z_j$: transaction $z_i$ and $z_j$ read the same tuple from Emp such that Emp.no = E1. Transaction $z_i$ updates the salary of E1 with the new value 90,000. After that transaction $z_j$ also updates the salary with the new value 80,000. Consequently, the update of transaction $z_i$ is lost because transaction $z_j$ overwrites it.

- The inconsistent analysis problem

This problem occurs whenever a retrieval transaction reads one attribute value of a tuple before another transaction updates it and reads another attribute value of the same tuple after the same transaction has updated it. That is, the retrieval only sees some of the update transaction’s result.

Example 4.2 Assume that employee E1 and E2 want to donate 10 percent of their salary to children who suffer from heart disease. Assume that transaction $z_i$ calculates the donation of E1 and E2, and transaction $z_j$ prints the sum donated by E1 and E2.

Transaction $z_i$

Read from Emp such that Emp.no = E1
Update Emp.donation := Emp.sal x 0.1
Update Emp.sal := Emp.sal – Emp.donation
Read from Emp such that Emp.no = E2
Update Emp.donation := Emp.sal x 0.1
Update Emp.sal := Emp.sal – Emp.donation

Transaction $z_j$

Read from Emp such that Emp.no = E1
4.2. Two main concurrency problems

\[ \text{Sum} := \text{Sum} + \text{Emp.donation} \]

Read from Emp such that Emp.no = E2

\[ \text{Sum} := \text{Sum} + \text{Emp.donation} \]

Output(Sum)

<table>
<thead>
<tr>
<th>EMP table</th>
<th>EMP.no</th>
<th>EMP.sal</th>
<th>EMP.donation</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>100,000</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>E2</td>
<td>120,000</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

The following shows an interleaved execution of the two transactions \( z_i \) and \( z_j \): transaction \( z_i \) reads and updates E1.donation. After that transaction \( z_j \) reads E1.donation and E2.donation and prints the sum of the donations. Finally transaction \( z_i \) reads and updates E2.donation. Consequently, transaction \( z_j \) prints the value 10,000, which is not the correct sum donated by E1 and E2. The error was caused by the interleaved execution of operations from different transactions. This interleaving causes transactions to interfere with each other, thereby leading to an inconsistent database.

One simple method to avoid these problems is to execute the transactions one at a time, called serial execution, i.e., not to allow transactions to be interleaved at all. It is easy to see that such serial executions are correct, because each transaction individually is correct and it will preserve consistency and leaves a consistent state for the next transaction to act upon. Of course the initial database state must also be consistent. But such serial executions lead to poor performance by not taking advantage of possible concurrency. On the other hand, arbitrary concurrency in the execution of transactions as shown in the above examples can destroy consistency. Much research has been devoted to increasing the efficiency of databases by allowing several transactions to execute concurrently while preserving consistency. The following section describes the concept of serializability that permits maximum concurrency while preserving consistency.
4.3 Serializability

Serializability is a correctness criterion for concurrency control. A concurrent execution of transactions is serializable if it is equivalent to some serial execution of the same transactions. This notion of equivalence to a serial execution is referred to as serializability, and is accepted as the cornerstone of database correctness.

Serializability theory is a tool to prove whether or not the concurrent execution of transactions is correct. In the theory, a history represents the structure of a concurrent execution of transactions and indicates the order in which the operations of the transactions were executed relative to each other. A history is serial if, for every two transactions in the history, all operations of one transaction appear before all operations of the other transaction i.e., there is no interleaving of operations of different transactions in the history. A history is called serializable if it represents a serializable execution.

There are two types of serializable executions: one is known as a view serializable execution [BSW79, Pap79], the other is known as a conflict serializable execution [BHG87]. Both types of serializable executions are outlined in the following definitions.

**Definition 4.1 Conflict serializability**

- Two histories are conflict equivalent if
  - both histories represent the structure of concurrent execution of the same transactions and,
  - conflicting operations of un aborted transactions appear in the same order in both histories.

- A history is conflict serializable if it is conflict equivalent to a serial history

**Example 4.3** From the example shown in the inconsistent analysis problem, consider a history representing the structure of concurrent execution of two transactions \(z_i\) and \(z_j\) as follows:

- Firstly, transaction \(z_i\) reads from Emp such that Emp.no = E1 and updates the value of E1.donation with 10,000.
4.3 Serializability

- Secondly, transaction $z_j$ reads the value of $E1$.donation and calculates $\text{Sum}$ such that $\text{Sum} = \text{Sum} + 10,000$.

- Thirdly, transaction $z_i$ reads from $\text{Emp}$ such that $\text{Emp.no} = E2$ and updates the value of $E2$.donation with 12,000.

- Fourthly, transaction $z_j$ reads the value of $E2$.donation and calculates $\text{Sum}$ such that $\text{Sum} = \text{Sum} + 12,000$, and prints the value of $\text{Sum}$.

This history is conflict equivalent to a serial history where all operations of $z_i$ appear before all operations of $z_j$. Thus this history is conflict serializable.

Definition 4.2 View serializability

- Two histories are view equivalent if
  - both histories represent the structure of concurrent execution of the same transactions and,
  - they have the same read-from relationship and,
  - the same final write operations appear in both histories.

- A history is view serializable if it is view equivalent to a serial history.

Example 4.4 Consider a history representing the structure of concurrent execution of three transactions $z_i$, $z_j$, and $z_k$ as follows:

- Firstly, transaction $z_i$ changes the value of data object $x$ to 1.

- Secondly, transaction $z_j$ changes the value of data object $x$ to 2.

- Thirdly, transaction $z_k$ changes the value of data object $x$ to 3.

- Fourthly, transaction $z_j$ changes the value of data object $y$ to 2.

- Finally, transaction $z_i$ reads the value of data object $y$.

This history is view equivalent to a serial history that has the following execution order.

- Firstly, transaction $z_j$ changes the value of data object $x$ to 2.
• Secondly, transaction $z_j$ changes the value of data object $y$ to 2.

• Thirdly, transaction $z_i$ changes the value of data object $x$ to 1.

• Fourthly, transaction $z_i$ reads the value of data object $y$.

• Finally, transaction $z_k$ changes the value of data object $x$ to 3.

Even though this history is a view serializable history, it is not a conflict serializable history, because in any equivalent serial history, all operations of transaction $z_i$ must appear before all operations of transaction $z_j$ and at the same time all operations of $z_j$ must appear before all operations of transaction $z_i$. There is no serial history satisfying such a condition. As a result, view serializable histories are more inclusive than conflict serializable histories. But in practice, it is too expensive to enforce view serializability of execution histories because their recognition is NP-complete [Pap79].

### 4.4 Extended transaction model

The traditional transaction model defines a transaction as a unit of concurrency control. This model does not provide much flexibility or high performance when used for complex applications such as long-lived transactions or distributed systems. As a solution to these limitations of the traditional transaction model, Moss proposed a nested transaction model.

Nested transactions are an extension of the traditional transactions, allowing the composition of transactions, concurrency within a transaction, and more graceful response to failure [Mos81]. Such extension allows for the dynamic decomposition of a transaction into a hierarchy of sub-transactions (or internal transactions), thereby preserving all properties of a transaction as a unit and assuring atomicity and isolation execution for every individual internal transaction [HR93].

The benefit of nested transactions is that they can be used to localize the effects of failures and faults in the system. When an uncommitted sub-transaction aborts, it can be rolled back without any side effects to other transactions outside its hierarchy. In the traditional transaction model, if part of a transaction fails due to faults in the system,
the entire transaction is undone to ensure atomicity. This advantage makes nested transactions highly desirable for distributed systems and long-lived transactions. The other benefit of nested transactions is that this model permits modular and concurrent composition of transactions. Sub-transactions facilitate a simple and safe composition of a program whose module may be designed and implemented independently.

### 4.5 Concurrency control algorithms

The majority of existing database concurrency control algorithms fit into one of two categories of locking [KS80, EL90], or non-locking strategies [BHG87].

The class of locking algorithms enforces the correctness of the concurrent execution of database transactions by imposing locks on data items and by enforcing specialized strategies to acquire and to release the locks. For example, the two-phase locking (2PL) algorithm [EGLT76] is the best known and probably the most frequently used in practice representations of the locking approach. Since its development many improvements have been proposed, such as strict two-phase locking [BG81], two-version two-phase locking [BP93], altruistic locking [SGA87, SGS94], and locks with constrained sharing [EL90].

On the other hand there is a class of algorithms that control the concurrent access to shared data by means other than locking. It includes time-stamp ordering [Ree78, BG81], serialization graph testing [Bad79, Cas81], versioning [AB87], and optimistic concurrency control [KR81]. The following two sub-sections provide a brief overview of two-phase locking and serialization graph testing mechanisms, respectively.

#### 4.5.1 Two-phase locking (2PL)

One way to ensure serializability is to require that access to data objects is available in a way which is mutually exclusive; that is, while one transaction accesses a data object, no other transaction can modify that data object. The most common mechanism used to implement this is locking. Locking based approaches, such as two-phase locking, are based on the notion of conflicts between database operations. In other word,
these approaches translate the non-conflicting and conflicting relationships between operations into shared and non-shared relationships between locks.

The 2PL approach [EGLT76] is the best known and probably the most frequently used in practice representation of the locking mechanism. In order to guarantee serializable execution of transactions, locking and unlocking must follow the two-phase locking rule. This rule requires that a transaction may not obtain any more locks after it has leased a lock on a data object. Most implementations of 2PL use the policy of releasing all locks when transactions commit or abort. This approach is known as a strict 2PL approach.

Simplicity of implementation and a lack of cascading aborts are the main advantages of the locking approaches. However the locking algorithms impose stronger limitations on the concurrent execution of database transactions by abolishing some of the conflict serializable histories. Moreover, the locking approaches are vulnerable to deadlocks [BHG87], such that for two transactions \( z_i \) and \( z_j \), transaction \( z_i \) requests a lock on a data object which is held by transaction \( z_j \) at the same time transaction \( z_j \) requests a lock on another data object which is held by transaction \( z_i \). An existing solution to this problem is based on the construction and run-time maintenance of a so-called wait-for graph [CKK90]. However, this requires detection of cycles in the wait-for graph, and it leads to similar problems with time complexity as detection of cycles in the serialization graph.

### 4.5.2 Serialization graph testing (SGT) mechanism

The serialization graph testing mechanism is able to directly schedule any operation in serializable execution history [EGLT76, BSW79, Pap79]. This is the main reason why, at least in theory, such a mechanism should provide the best performance.

The serialization graph testing (SGT) mechanism maintains a serialization graph (SG). The SG is a directed graph such that \( SG = (N,E) \) where \( N \) represents a set of transactions called nodes, and \( E \) denotes a set of pairs of nodes called edges.

A node \( z_i \) for transaction \( z_i \) is added to \( N \) when an operation \( p_i \) issued by \( z_i \) is sent for execution, if the node \( z_i \) does not exist in \( N \). An edge \([z_i,z_j]\) is added to \( E \) when an
operation \( p_j \) issued by \( z_j \) is executed, if the operation \( p_j \) conflicts with an operation that has been earlier executed by uncommitted transaction \( z_i \).

The construction of SG is a simple method for checking whether the execution history is serializable or not. If the resulting SG contains a cycle when an edge \([z_i, z_j]\) is added to \( E \), then the resulting execution history is non-serializable. The non-serializable execution history may cause cascading aborts such that all transactions which have incoming edges from \( z_j \) are aborted and aborting these transactions may trigger further abortions.

In practice, the complexity of implementation and the cascading abort phenomenon significantly reduces their efficiency [ACL87]. The main problems are in the real time maintenance of the serialization graph and in the testing of execution history. Conflict serializability testing requires the detection of cycles in a serialization graph. The fastest known algorithms [CLR90] or [IRW93] run in \( O(m \times n) \) time where \( m \) is the number of edges and \( n \) is the number of nodes, or alternatively in \( O(n^{2.38}) \) time using a fast binary matrix multiplication [CW90].

4.6 Hybrid concurrency control in database systems

This section presents a hybrid approach to the construction of concurrency control algorithms for database systems. A more detailed description of hybrid concurrency control is needed because the rest of this thesis is based on this technique. It is based on [KG96].

The hybrid approach generalizes earlier works on locking and non-locking concurrency control strategies [Grub94, WFGL88]. The hybrid approach means that it has the ability to dynamically adjust the behaviour of concurrency control algorithms to changing environmental parameters.

The family of algorithms presented in the hybrid approach enforces the conflict serializable execution of database transactions by a mixture of locking and non-locking strategies. The impact of both approaches on the specific algorithms is measured by
the largest admissible size of cascading aborts and the total number of transactions that concurrently run with a particular transaction. It is proved that for extreme values of parameters, this strategy reduces itself either to a strict two-phase locking approach or serialization graph testing.

This section discusses the selected implementation aspect of the hybrid concurrency control technique. Sub-section 4.6.1 provides basic concepts of the hybrid approach. Sub-section 4.6.2 provides the basic definition and the notational conventions used throughout this section. The specification of the hybrid scheduling algorithm is given in sub-section 4.6.3. Sub-section 4.6.4 discusses the concept of the $H(n,k)$ algorithm. Finally, Sub-section 4.6.5 offers a conclusion.

4.6.1 Basic concept of hybrid approach

The classification of concurrency control algorithms depends on the kind of action a scheduler undertakes when it receives an operation that introduces a risk of violation of conflict serializable execution. For instance, this occurs when a transaction attempts to access, in conflicting mode, a data item that has been accessed earlier by another transaction. Consequently, the scheduler may delay execution of an operation until the risk of non-serializable execution disappears. Such behaviour is typical of locking algorithms. The other possibility is to risk non-serializable execution and to schedule the operation for immediate execution. This approach is typical of non-locking strategies. In the hybrid approach the scheduler may either delay an operation or it may submit it for execution. The decision as to what kind of action should be carried out at a given moment in time depends on the present state of the system. For instance, in order to work out the decision, the scheduler may use information about frequency of access to data items carried out in a certain period of time by different transactions.

In the hybrid approach, the decision as to whether an operation should be executed or delayed depends on a pair of parameters $(n,k)$. The first one determines the "maximum size" of cascading abort. The maximum size of cascading abort means the total number of transactions that should be abnormally terminated if any other transaction running in the system fails or if it must be abnormally terminated by the system. The
second parameter determines the maximum number of active transactions that any other active transaction depends on. It is said that an active transaction depends on another active transaction if the first one accesses, in conflicting mode, a data item that had been earlier accessed by the other one. The decision as to whether the execution of a database operation should be delayed depends on the current state of the system which is represented by the parameters $n$, and $k$. If the new inter-transaction dependencies that arise after execution of a database operation violate one of the parameters, then execution of such an operation must be postponed by the scheduler. Otherwise, if an operation does not violate conflict serializability, then it can be scheduled for immediate execution. In fact, such an approach represents the entire family of concurrency control algorithms. By taking into consideration the specific values of parameters $n$, and $k$, different concurrency control strategies are obtained. For instance, when $n = 0$ and $k = 0$, the hybrid strategy reduces to classical strict two-phase locking. When $n > 1$ and $k = 1$, the hybrid strategy reduces to altruistic locking. If $n \to \infty$ and $k \to \infty$ the strategy is equivalent to serialization graph testing. For the specific values of the parameters $n,k$, the hybrid approach preserves all advantages of locking approaches and it also accepts more conflict serializable histories, improving the overall performance of the system.

4.6.2 Notations and definitions

This sub-section provides the basic definitions and notations that are used through this chapter.

**Definition 4.6.1** An execution history is a sequence of entries $p_1(x), \ldots , p_n(x)$, where $p_i(x)$ denotes an operation $p$ issued by transaction $z_i$ operates on a data item $x$, and $p \in \{\text{read, write}\}$.

**Definition 4.6.2** A pair of database operations, issued by different transactions, conflicts if both of them access the same data item and at least one of them is a write operation.

**Definition 4.6.3** The execution history is conflict serializable if a serial execution history of the same transactions exists such that in both executions the order of conflicting
operations is exactly the same [EGLT76, BSW79, Pap79].

**Definition 4.6.4** An extended serialization graph, ESG, is a triple such that $ESG = (N, E_s, E_d)$ where $N$ is a set of nodes labelled with non-committed transaction identifiers, $E_s$ is a set of "solid" edges such that $E_s \subseteq T \times T$ and $E_d$ is a set of "dashed" edges such that $E_d \subseteq T \times T$.

If an edge $[z_i, z_j] \in E_d$ then it means that transaction $z_j$ submitted an operation $p_j$ for execution, but the operation has not executed yet, and that the operation conflicts with an operation $p_i$ that has been executed earlier by $z_i$. When the operation $p_j$ is scheduled for execution, the dashed edge $[z_i, z_j]$ should be changed into a solid edge. If an edge $[z_i, z_j] \in E_s$ then transaction $z_j$ executed an operation that conflicts with the other operation that has been executed earlier by transaction $z_i$.

**Definition 4.6.5** Transaction $z_j$ directly depends on transaction $z_i$ if $z_j$ executed an operation $p_j(x)$ after transaction $z_i$ executed an operation $p_i(x)$, and $p_j$ conflicts with $p_i$ (i.e. for an edge $[z_i, z_j] \in E_s$, $z_j$ directly depends on $z_i$).

**Notation 4.6.1** A set of transactions that directly depends on transaction $z_i$ is denoted by $d(z_i)$.

**Definition 4.6.6** Transaction $z_i$ is a direct predecessor of transaction $z_j$ if $z_j \in d(z_i)$.

**Definition 4.6.7** A set of transactions $d^*(z_i)$ that depends on $z_i$ is recursively defined as follows.

$$d^*(z_i) = \{ z_j | z_j \in d(z_i) \text{ or there exists } z_k \text{ such that } z_k \in d^*(z_j) \text{ and } z_j \in d(z_k) \}.$$

**Definition 4.6.8** A set of transactions that are direct predecessors of $z_j$ is denoted by $p(z_j)$.

**Notation 4.6.2** For an edge $[z_i, z_j]$, $z_i$ is called a from-node and $z_j$ is called a to-node. $|d(z_i)|$ denotes a cardinality of the set $d(z_i)$. 
Definition 4.6.9 A $H(n,k)$ concurrency control algorithm is defined as a strategy where for any transaction $z_i$,

1. $\|d^*(z_i)\| \leq n$, and
2. $\|p(z_i)\| \leq k$

The parameter $n$ of $H(n,k)$ algorithm determines the "acceptable size" of cascading abort. The parameter $k$ determines the number of transactions that a given transaction depends upon. Both conditions, $\|d^*(z_i)\| \leq n$ and $\|p(z_i)\| \leq k$, should be satisfied at every moment in time. For instance, in $H(2,k)$ algorithm, a failure of any transaction in any moment of time cannot trigger the failures of more than two other transactions.

The extended serialization graph given in Figure 4.1 represents the conflicts in an execution history $[r_1(x), w_2(x), w_3(x)]$. Figure 4.2 represents this after the submission of operation $r_4(x)$, where $n = 2$, $k = 2$ of $H(n,k)$.

Figure 4.1: A sample serialization graph

Figure 4.2: A sample extended serialization graph
4.6.3 Scheduling algorithms

This sub-section contains specification of the scheduling algorithms that enforce the hybrid approach. The following typical architecture of the transaction management system based on a serialization graph testing principle is adopted.

The following picture is an architecture of a database system.

![Architecture of hybrid database system model](image)

The scheduler receives an operation from one of the active transactions, recognizes it and then determines whether its execution violates the conflict serializability criterion. If after modification a serialization graph does not contain a cycle then the scheduler submits an operation for execution, otherwise it rejects the operation and determines what transaction should be adopted to avoid non-serializable execution.

In $W(n,k)$ algorithm, the scheduler passes an operation to a conflict analyzer to find all transactions that executed at least one operation that conflicts with the given one. Then the set of conflicts is passed to the serialization graph analyzer to test serializability and the two conditions of $W(n,k)$ algorithm. If the extended serialization graph has a cycle composed of any type of edges then the operation is rejected. Otherwise, if the two conditions of $H(n,k)$ algorithm are satisfied for each conflict that has been identified earlier, the scheduler submits the operation for execution and appends the new solid edge to the extended serialization graph. If the two conditions of $H(n,k)$ algorithm are not valid the scheduler delays execution of the operation and appends a new dashed edge to the extended serialization graph. When a delayed operation can be executed the scheduler replaces the respective dashed edges with solid edges.
4.6. Hybrid concurrency control in database systems

The scheduler co-ordinates the actions of the conflict analyzer and serialization graph analyzer in the way described by Algorithm 4.6.1.

**Algorithm 4.6.1 Scheduler**

*Input:* Operation \(p_j(x)\) submitted by transaction \(z_j\), where \(p_j \in \{\text{read, write, commit, abort}\}\).

*Output:* Operation \(p_j\).

*Method:*

- *If* \(p_j \in \{\text{read, write}\} \) *then* the scheduler submits it to the conflict analyzer (Algorithm 4.6.2). The conflict analyzer returns a set \(C\) of conflict pairs. If \(C\) is empty then the scheduler appends \(p_j\) to the execution history and sends it to the data manager for execution. Otherwise, the scheduler sends the set \(C\) to the serialization graph analyzer (Algorithm 4.6.3). The serialization graph analyzer returns one of the following responses:

  1. **Execute:** Scheduler appends \(p_j\) to execution history and sends \(p_j\) to data manager for execution.
  2. **Delay:** Scheduler appends \(p_j\) to its delay queue.
  3. **Reject:** Scheduler submits abort actions for transaction \(z_j\) into its input queue.

- *If* \(p_j = \text{commit}\) *then* the scheduler performs the following actions:

  1. If there exists transaction \(z_i\) such that \(z_j \in d(z_i)\) then the scheduler inserts \(p_j\) into its delay queue.
  2. Otherwise, the scheduler executes all commit actions and for each transaction \(z_k\) such that \([z_j,z_k] \in E_d\), it searches its delay queue for operation \(p_k\). If any operation \(p_k\) is found the scheduler inserts it into its input queue and removes the edge \([z_j,z_k]\) from \(E_d\).

- *If* \(p_j = \text{abort}\) *then* the scheduler executes all abort actions for transaction \(z_j\) and for each transaction \(z_k\) such that \(z_k \in d^*(z_j)\) it submits abort operation into its
input queue. In $H(n,k)$ algorithm, there should be no more than $n$ such transactions. For execution $z_n$ such that $[z_j, z_n] \in E_d$, the scheduler extracts operation $p_n$ from its delay queue and submits $p_n$ into its input queue.

Algorithm 4.6.2 Conflict Analyzer

Input: Operation $p_j[x]$.  
Output: A set $C$ of pairs $\{ [z_1, z_j], \ldots, [z_k, z_j]\}$ such that $z_j \in d(z_1), \ldots, z_j \in d(z_k)$.  
Method:

The conflict analyzer receives an operation $p_j(x)$ from the scheduler and finds all transactions that have executed operations conflicting with $p_j(x)$. For each transaction $t_k$ found, the conflict analyzer constructs a pair $[z_k, z_j]$ and adds it to the output set $C$.

Algorithm 4.6.3 Serialization Graph Analyzer

Input: A set $C$ of pairs $\{ [z_1, z_j], \ldots, [z_k, z_j]\}$ such that $z_j \in d(z_1), \ldots, z_j \in d(z_k)$.  
Output: Updated extended serialization graph and decision whether operation should be executed, delayed, or rejected.  
Method:

For each pair $[z_i, z_j]$ from the input set, serialization graph analyzer performs the following actions:

1. Serialization graph analyzer tests the two conditions of $H(n,k)$ algorithm i.e., $\| d^*(z_j) \| \leq n$, and $\| p(z_j) \| \leq k$. If the two conditions are satisfied, the analyzer adds a new solid edge $[z_i, z_j]$ to the extended serialization graph. Otherwise, it adds a dashed edge $[z_i, z_j]$ to the graph.

2. If after addition of a solid or dashed edge $[z_i, z_j]$, the extended serialization graph has a cycle, then the serialization graph analyzer returns a reject value and quits its actions.

3. Otherwise, the analyzer returns execute when a solid edge occurs, or the analyzer returns delay when a dashed edge occurs.

4.6.4 $H(n,k)$ algorithms

This section provides a brief overview of $H(n,k)$ algorithms.
• H(0,k) algorithm (where k > 0)

The serialization graph analyzer of H(0,k > 0) algorithm has the same properties as the serialization graph analyzer of H(0,0) algorithm. For a pair \([z_i,z_j]\) received from the conflict analyzer, if the serialization graph analyzer of H(0,k > 0) algorithm allows \(z_j\) to have a direct predecessor, there should be at least one transaction (i.e., \(z_j\)) to be aborted by cascading phenomenon when the direct predecessor of \(z_j\) fails. This violates the assumption that there are no transactions to be aborted by the cascading phenomenon when \(n\) equals zero. Thus, in the case of H(0,k > 0), there is no transaction that depends on any other in the extended serialization graph.

• H(0,0) algorithm

The serialization graph analyzer of H(0,0) algorithm returns a delay to the scheduler whenever it receives a pair \([z_i,z_j]\) from the conflict analyzer, since in the H(0,0) algorithm, the condition \(|d^*(z_i)| \leq 0\) is not always satisfied. Therefore all transactions should be delayed if operations issued by those transactions for execution conflict with an operation executed already by the other transaction and if any cycles are not detected when the conflict edges are appended to the set of dashed edges.

The H(0,0) algorithm has a similar property to that of the strict two-phase locking algorithm, since the strict two-phase locking algorithm requires a scheduler to release all locks of a transaction together when the transaction commits. When a transaction \(z_j\) wants to have a lock that a transaction \(z_i\) holds, the transaction \(z_j\) must waits until transaction \(z_i\) commits. It has the same meaning that the execution of transaction \(z_j\) is delayed by transaction \(z_i\) as the H(0,0) algorithm. Both algorithms (strict two-phase locking algorithm and H(0,0) algorithm) avoid the cascading aborts phenomenon. To detect conflict operations, the strict two-phase locking algorithm uses a lock-list for data items, and the H(0,0) algorithm uses a read and write set of data items for active transactions. But both algorithms require the same space to detect conflict operations. The differences between a strict two-phase locking algorithm and a H(0,0) algorithm are as follows;
4.6. Hybrid concurrency control in database systems

1. The former is vulnerable to the dead-lock problem [BH87], but the latter is dead-lock free, since the former uses a locking mechanism, but the latter uses an extended serialization graph which is composed of a serialization graph and a wait-for graph.

2. When the former uses a wait-for graph to solve the dead-lock problem, the algorithm testing cycles for the wait-for graph has a complexity $O(nm)$, where $n$ is the number of nodes and $m$ the number of edges in the graph, but the latter has a complexity of $O(n/2)$ [KG96].

- H(1,k) algorithm (where $k > 1$)

When the serialization graph analyzer of H(1,k) algorithm receives a pair $[z_i, z_j]$ from the conflict analyzer, the H(1,k) algorithm acts as follows.

Given a pair $[z_i, z_j]$, $z_j$ should wait for $z_i$ if there is a transaction depending on $z_j$ or the number of direct predecessors of $z_j > k$, or a direct predecessor of $z_i$ exists. Otherwise $z_j$ directly depends on $z_i$.

- H(1,1) algorithm

The H(1,1) algorithm is similar to the H(1,k) algorithm except that in H(1,1) algorithm, any transaction in the system has only one direct predecessor. In H(1,k) algorithm, the set of direct predecessors of a transaction may have at most $k$ elements, but in H(1,1), the set of direct predecessors of any transaction has only one element.

- H($n \to \infty$, $k \to \infty$) algorithm

The extended serialization graph of H($n \to \infty$, $k \to \infty$) algorithm is the same as the serialization graph, since there are no dashed edges in the extended serialization graph. The main concept of the H($n \to \infty$, $k \to \infty$) algorithm is to represent the extended serialization graph as an adjacency matrix. Let $G$ be an extended serialization graph with nodes $N_1, N_2, \ldots, N_n$. The adjacency matrix of $G$ is the $n \times n$ square matrix $M = (N_{ij})$, where $N_{ij} = s$ if an edge from $N_i$ to $N_j$
4.6. Hybrid concurrency control in database systems

(later on \((N_i, N_j)\)) exists. Given an edge \((N_i, N_j)\), the matrix \(M\) is constructed as follows:

1. Put \(s\) on \(N_{kh}\), for all edges \((N_k, N_h)\) such that \((N_k, N_h) \in \{(N_s, N_a) | N_s \in SN^* \text{ and } N_a \in AN^*\}\)
   where \(SN^* = \{N_t | N_{ti} = s\} \cup \{N_i\}\) and \(AN^* = \{N_v | N_{jv} = s\} \cup \{N_j\}\).
2. If an element in the main diagonal of \(G\) has the value \(s\) then a cycle is detected.

4.6.5 Conclusions

A new family of concurrency control algorithms was introduced. It uses both locking and non-locking strategies to achieve better performance of concurrently running database transactions. The algorithm determines the maximum size of a cascading abort and the total number of transactions that execute concurrently with a particular transaction to decide whether an operation that conflicts with another one may be immediately executed or if it should be delayed.

This approach generalizes previous work on locking and non-locking concurrency control strategies and provides algorithms that share the properties of both of them. Another important advantage of algorithms is their ability to dynamically adjust their behaviour to the changing parameters of the environment in which they are working.

Depending on the parameters, hybrid algorithms are able to change their properties from a locking-oriented approach to non-locking based strategies. It has been proved that the \(H(0, 0)\) algorithm reduces the strict two phase locking algorithm, but \(H(0, 0)\) algorithm did not suffer from a dead-lock problem and it has a lower complexity \(O(n/2)\) than \(O(nm)\).

It has also been proved that the \(H(1,k)\) algorithm enables more transactions to execute concurrently than the \(H(0,0)\) algorithm, while it has the same complexity \(O(n/2)\) as \(H(0,0)\) algorithm.
It showed that the $H(\infty, \infty)$ algorithm reduces to a serialization graph testing algorithm. $H(\infty, \infty)$ algorithm uses an adjacency matrix to represent an extended serialization graph, and it has a complexity of $O(n^2)$.

As a result, the large values of the parameters provide algorithms that are less restrictive in the detection of serializable histories but, on the other hand, require more time for the analysis of additional data structures and increase the risk and size of a cascading abort.
Part III

Hybrid Concurrency Control for Prioritized Rules in Active Database Systems
Chapter 5

Introduction

This part introduces a new transaction management protocol that allows not only for deterministic but also terminated execution of rules in active database systems.

In active database system, rule processing occurs as a result of arbitrary database changes: certain rules are triggered initially, and their execution can trigger other rules or the same rules and so on-conceivably resulting in a finite set of triggered rules.

During the rule processing, it can be very difficult to predict how rules in the finite set will behave. Because the same finite set of rule transactions may not only behave differently when considered in different orders, yielding unexpected results (confluence problem), but also produce infinite cyclic executions (termination problem).

First, to solve the confluence problem, a strategy is introduced that eliminates the situation where different executions of rules in the finite set produce different results. This strategy is based on the assumption that there should only be one execution semantic for a given system of rules in the finite set. A unique execution semantic can be easily enforced by a rule designer by associating with each rule a priority that determines a global execution order. Unfortunately, efficient implementation of such a system is much more complicated. A property that execution of rules in the finite set is equivalent to execution of respective database transactions suggests a solution based on one of the transaction management techniques. In order to have unique execution semantics for a given system of rules in the finite set, the strategy enforces a unique serialization order of the respective database transactions. Such a conclusion leads to a transaction management protocol that enforces serializable execution of database transactions such that the respective serialization order is consistent with an order determined by the rule priorities.
A naive solution to this problem would be a serial execution of transactions in the same order as an order enforced by the priorities of respective rules. Of course this thesis looks for a more efficient solution when transactions are concurrently running in the system. The problem is not in enforcing any serializable execution, but in enforcing an execution consistent with a serialization order determined by the priorities of rules. To solve this problem, the thesis uses a transaction management technique based on serialization graph maintenance and testing. The serialization graph maintenance and testing allows any interleaving of operations that is serializable. In this sense, it is more lenient than time-stamp ordering and two-phase locking, since time-stamp ordering only allows timestamp ordered executions of operations and two-phase locking doesn’t allow certain interleaving of operations.

Secondly, this thesis solves the termination problem. The execution of the action part of a rule may generate new events that may trigger other rules and so on, the result of this might be an infinite cyclic execution of some rules.

In this thesis, a rooted graph is used to detect the infinite cyclic execution of rules. The rooted graph is a directed graph, where a node is distinguished from the others and is called the root node. The root node denotes an event which triggers rules initially. The others denote a finite set of rules. There is an edge labelled $e_i$ from a node $r_i$ to another node $r_j$ if $r_j$ is triggered by $e_i$ which is associated with command $c_i$ in the action part of $r_j$.

The infinite cyclic execution of rules is detected if the resulting rooted graph contains a cycle. This is based on the assumption that the execution of a finite set of rules triggered by an event is deterministic.

**Example 5.1 Deterministic rule execution**

For three rules $r_i$, $r_j$ and $r_k$, assume that the action of $r_k$ generates events which trigger $r_i$ and $r_j$, and the action of $r_j$ generates events which trigger $r_k$.

- **Deterministic rule execution**
  
  Two rule $r_i$ and $r_j$ are triggered if $r_k$ executes and generates events. Two possible cyclic executions are detected:

  - In case the execution order between $r_i$ and $r_j$ is that $r_j$ should follow $r_i$:
\[ r_k \rightarrow r_i \rightarrow r_j \rightarrow r_k, \]

- In case the execution order between \( r_i \) and \( r_j \) is that \( r_i \) should follow \( r_j \):
  \[ r_k \rightarrow r_j \rightarrow r_i \rightarrow r_k, \]
  where \( \rightarrow \) denotes an execution order such that if \( r_k \rightarrow r_i \), then \( r_k \) executes before \( r_i \).

- Non-deterministic rule execution

  If there is no execution order between rules involved in a cyclic execution then the cyclic execution may repeat indefinitely such as
  \[ r_k \rightarrow r_i \rightarrow r_j \rightarrow r_k \rightarrow r_j \rightarrow r_k \rightarrow r_i \rightarrow r_j. \]

In general, two types of infinite cyclic executions may happen during the rule processing: one produces an infinite number of possible database states, the other produces infinite repetition of the database states. The former could be prevented if a rule designer gives some restriction to the action which causes the cyclic executions. The latter is detected when the action which causes the cyclic executions is executed on a database state such that it must be repeated more than once. This thesis introduces a strategy to test whether or not an action of a rule involved in the cyclic executions is executed repeatedly on the same database state.

Thirdly, this thesis suggests a solution on the question: what should be done when a transaction attempts to access a data object in a mode that can potentially violate a given serialization order? My solution is to replicate both data and processes. Replication of data is a well known technique that requires a multiversion concurrency control mechanism. Replication of processes is achieved by transaction cloning. The idea is explained in the following basic example. Assume that according to a given serialization order transaction \( z_j \) should follow a transaction \( z_i \). As shown in figure 5.1, there exist three cases when \( z_j \) may violate the order.

The first one is when transaction \( z_j \) executes Write(\( x \)) operation and later on transaction \( z_i \) executes Read(\( x \)) operation. To solve this problem it is enough to create a new copy (version) of data item \( x \) when \( z_j \) writes it. The same strategy can be applied to the second case when \( z_j \) executes Write(\( x \)), later on \( z_i \) executes Write(\( x \)). Versioning is not sufficient in the third case i.e., when \( z_j \) reads data item \( x \) and \( z_i \) may write data
To solve this problem a clone transaction $z_k$ of transaction $z_j$ is created if the probability of Write(x) to be executed in the future is higher than a given threshold. The clone transaction $z_k$ of $z_j$ is a transaction that has an identical execution history and it is in the identical state as $z_j$ when it attempts to read item x. Transaction $z_j$ is suspended before read(x) and only its clone transaction continues execution. If a clone is involved in execution that violates a given serialization order it is aborted and transaction $z_j$ resumes its execution from a point cloning. If clone transaction $z_k$ is able to reach a commit point and successfully commit itself then transaction $z_j$ disappears from the system without either abort or commit. In this strategy a clone transaction plays the role of a reconnaissance transaction that performs the actions in advance of its prototype. Of course, whenever a clone transaction attempts to execute an operation that may violate a given serialization order, it can be cloned as well. Transaction cloning is especially useful when the system deals with a transaction that represents a long sequence of actions of the ECA rules and when it is too expensive to re-execute the entire transaction when it violates a given serialization order.

This thesis also introduces the concept of compensation operation as another way in which a serialization order violated by a clone transaction can be restored. The idea comes from an assumption that rules are executed at a set level granularity.
Assume that a clone transaction is created when its cloning transaction $z_i$ attempts to execute an operation $p_i$ which may violate a given serialization order in the future. If the violation occurs, the operation $p_i$ should be re-submitted by the cloning transaction $z_i$ for execution again. It is a waste of time for the operation $p_i$ submitted by $z_i$ to read tuples which the clone transaction has read. To avoid this problem, a compensation operation is executed instead of $p_i$ re-submitted by $z_i$. The compensation operation is an operation which is executed on the tuples updated by the clone transaction, and produces the same results as if $p_i$ submitted by $z_i$ executes on entire tuples. Compensation operations are useful when a transaction consists of a sequence of independent operations that can be individually compensated.

Finally, a hybrid approach is introduced because the predefined execution order of rules can be bounded in a serialization graph. Therefore it is possible to dynamically adjust the behaviour of concurrency control algorithms to changing environmental parameters. It is proved that for extreme values of parameters, the hybrid strategy reduces itself either to a locking approach or to a non-locking approach.
Chapter 6

Basic Concepts

This chapter provides definitions of basic concepts and introduces some notational conventions.

6.1 Commands vs atomic operations

As mentioned earlier, the capital C denotes a set of commands such that:
\[ C = \{(I, t_i, t) \mid t_i \in DB\} \cup \{(D, t_i, \psi) \mid t_i \in DB\} \]
\[ \cup \{(U, t_i, a_k, \psi) \mid t_i \in DB\} \cup \{(R, t_i, \psi, t_j) \mid t_i \in DB\}, \]
where \( \psi \) is a tuple filter. It is a Boolean expression that identifies the tuples to be deleted, to be updated or to be retrieved.

The Boolean expression is made up of a number of clauses of the form:

- \((t_i.a_i)\) (comparison op) (constant value) or

- \((t_i.a_i)\) (comparison op) \((t_i.a_j)\)

where (comparison op) is normally one of the operators \(\{=,\lt,\gt,\le,\ge,\ne\}\).

Assume that each command \(c_i\) in application programs or rules is translated into a command block containing a sequence of atomic operations.

Definition 6.1 Atomic operations are defined as follows:

- \((\text{start}, c_i)\) is used to begin executing the command block of \(c_i\).

- \((\text{end}, c_i)\) is used to indicate the termination of the command block of \(c_i\).

- \((\text{insert}, t_i, t)\) is used to insert tuple \(t\) into table \(t_i\).
• *(delete, t_i, t)* is used to delete tuple t from t_i.

• *(update, t_i.a_k, t)* is used to update the value of attribute a_k of tuple t in t_i.

• *(next, t_i)* is used to move a pointer in t_i that points to a tuple t to the next tuple of t.

• *(first, t_i)* is used to set a pointer to a position before the first tuple of t_i.

• *(get, t_i, t)* is used to read a tuple t indicated by a pointer in t_i.

The following implementation model shows how to translate a command c_i into the command block of c_i containing a sequence of atomic operations. Suppose that command c_i operates on table t_i.

**Implementation Model 6.1 Procedures:**

*In the case where c_i is a retrieval command (R, t_i, \psi, t_j):*

Procedure c_i begin

(start, c_i);

(first, t_i);

while not end of (t_i);

(get, t_i, t);

if \psi then save t in temporary table t_j;

(next, t_i);

end while;

(end, c_i);

end;

*In the case where c_i is a delete command (D, t_i, \psi):*

Procedure c_i begin

(start, c_i);

(first, t_i);

while not end of (t_i);

6.1. Commands vs atomic operations

\[(get,t_i,t)\];
\[\text{if } \psi \text{ then } (delete,t_i,t)\];
\[(next,t_i)\];
end while;
\[(end,c_i)\];
end;

In the case where \(c_i\) is an update command \((U,t_i.a_k,\psi)\):

Procedure \(c_i\) begin
\[(start,c_i)\];
\[(first,t_i)\];
while not end of \((t_i)\);
\[(get,t_i,t)\];
\[\text{if } \psi \text{ then } (update,t_i.a_k,t)\];
\[(next, t_i)\];
end while;
\[(end,c_i)\];
end;

Notation 6.1 An atomic operation is denoted by the letter \(p\) possibly followed by a subscript.

It is possible that the execution of the command block of \(c_i\) may attempt to change a database state in some way which would invalidate the command block of another command \(c_j\) while the command block of \(c_j\) is actually loaded and executing. It would be undesirable if such a change were allowed to become effective while the command block of \(c_j\) is in the middle of execution. To prevent this from occurring, a conflicting type between two command blocks is defined as follows:

Definition 6.2 A set of attributes of which the values to be read by atomic operations in the command block of \(c_i\) is denoted by \(R(c_i)\), such that \(R(c_i) = \{t_k.a_1, \ldots, t_k.a_n\}\),
where \( a_1, \ldots, a_n \in A(t_k) \). A set of attributes of which the values to be modified by atomic operations in the command block of \( c_i \) is denoted by \( W(c_i) \), such that \( W(c_i) = \{ t_k.a_1, \ldots, t_k.a_n \} \), where \( a_1, \ldots, a_n \in A(t_k) \).

**Definition 6.3** A command block of \( c_i \) issued by transaction \( z_i \) conflicts with a command block \( c_j \) issued by transaction \( z_j \) if \( c_i \) and \( c_j \) operate on the same table, \( W(c_j) \cap R(c_i) \neq \emptyset \) and the operation \((\text{start},c_j)\) of \( c_j \) executes before the operation \((\text{start},c_i)\) of \( c_i \).

### 6.2 Rules

For each rule \( r_i \), the condition-part and the action-part of rule \( r_i \) consists of commands. The commands are translated into respective command blocks by the translator during the compile-phase. Atomic operations in a command block are executed one after another in the order in which they appear in the sequence.

**Notation 6.2** \( R = \{r_1, \ldots, r_n\} \) denote a set of rules in the system.

**Definition 6.4** For a rule \( r_i \in R \), an event symbol in the event-part of \( r_i \) is any symbol from a set \( E \) of pairs such that \( E = \{(I,t_i),(D,t_i),(U,t_i,a_k)\} \).

**Notation 6.3** An event symbol is denoted by the letter \( e \) possibly followed by a subscript.

**Definition 6.5** The granularity of rule execution is defined as a set of tuples. An event \( e_i \) associated with a command \( c_i \) occurs after atomic operation \((\text{end},c_i)\) in the command block of \( c_i \) is executed.

**Definition 6.6** The condition for any rule is a predicate expressed as evaluation of a query; returns true if the result of the query is not empty.

**Definition 6.7** For two rules, \( r_i \) and \( r_j \), \( r_i \) has a precedence over \( r_j \) if \( r_i.o < r_j.o \), where \( r_i.o \) denotes the priority of rule \( r_i \).
6.3 Static analysis of rules

Rules in an active database system can be analyzed during compile time. Several static rule analysis models have been developed [WH95], [BCP95], [AWH92]. Goals of such analysis models are to detect potential loops between triggered rules. This section describes how to detect potential loops between triggered rules, and how to determine the potential execution order for triggered rules. To do this, this thesis assumes that for each event symbol $e_i$ in $E$, a rooted graph denoted by $G(e_i)$ is constructed by the static rule analysis. The $G(e_i)$ may have a finite set of triggered rules and a set of edges. A potential loop is detected if the resulting rooted graph contains a cycle.

**Definition 6.8** A rooted graph $G(e_i)$ is defined as a pair $(N,V)$, where

- one and only one of the nodes in $N$ is labelled by $e_i$ and
- the other nodes are labelled by rule identifies and
- if rule $r_k$ is triggered by event $e_j$ which is generated by $r_i$ then edge $[r_i,r_k] \in V$, and it is labelled by $e_j$.

The rooted graph $G(e_i)$ given in Figure 6.1 represents that the event $e_i$ of the root node of $G(e_i)$ triggers $r_i$ and $r_j$. The execution of the action part of rule $r_i$ generates a new event $e_j$ and it triggers rule $r_k$ and rule $r_n$.

In an active database system, a partial order defining the happens-before relationship between rules may specify the execution order of rules. This thesis assumes that the potential execution order of rules in a rooted graph $G(e_i)$ results from the static rule analysis, and it is based on a partial order given to rules as priorities.

It is difficult to determine whether or not a rule's condition is evaluated to true during the static rule analysis, since the condition evaluation depends on the state of the database at execution time. The word 'potential' is used because of the assumption that the conditions of rules are evaluated to true during the static analysis.

This thesis assumes that each command $c_i$ of rule $r_i$ has a probability, denoted by $\lambda$, that atomic operations in the command block of $c_i$ change a database state when the command block of $c_i$ is executed on the database state in the future.
6.3. Static analysis of rules

Definition 6.9 The execution order of commands of rule $r_i$.

A directed graph $O(r_i)$ is defined as a pair $(C, E)$, where $C$ is a set of commands of $r_i$, called nodes, and $E$ is a set of ordered pairs of nodes called edges. If a given execution order is that $c_j$ follows $c_i$ then edge $[c_i, c_j] \in E$, where $c_i$ and $c_j$ are commands of $r_i$. Each node has a probability which the respective command is executed in the future.

Figure 6.2 shows the execution order of commands of rule $r_i$, where for a node $c_k(i)$, the $i$ of $(i)$ denotes the probability of command $c_k$ to be executed in the future.
6.3. Static analysis of rules

Definition 6.10 A command \( c_j \) potentially conflicts with a command \( c_i \) if the following conditions are satisfied:

- they operate on the same table and
- they are in different rules and
- \( W(c_i) \cap R(c_j) \neq \emptyset \) and
- the priority of a rule containing \( c_i \) is less than another rule containing \( c_j \) and

This thesis defines \( P(e_i) \) as a directed graph representing the potential execution order of commands of rules in \( G(e_i) \) in the following way.

Definition 6.11 Given a rooted graph \( G(e_i) \), the potential execution order of commands of rules in \( G(e_i) \) is that \( P(e_i) = (M, W) \), where
6.3. Static analysis of rules

- \( M \) is a set of commands of rules in \( G(e_i) \), called nodes.

- \( W \) is a set of edges. If there is an edge from node \( r_i.c_i \) to node \( r_j.c_j \) labelled with \( \lambda \) in \( W \), then \( r_j.c_j \) potentially conflicts with \( r_i.c_i \), where \( r_i.c_i \) refers to the command \( c_i \) of rule \( r_i \), and \( \lambda \) denotes the probability of \( r_i.c_i \) to be executed in the future.

From Figure 6.1, assume that the priority of \( r_i \) is less than \( r_j \). Assume that the probability of \( r_i.c_i \) which changes a database state is \( \lambda_1 \) and the probability of \( r_k.c_k \) which changes a database state is \( \lambda_2 \).

If the E-CA coupling mode of \( r_i, r_j, r_k \) and \( r_n \) is an immediate mode and \( r_j.c_j \) potentially conflicts with \( r_i.c_i \) and \( r_k.c_k \) then the potential execution order of commands of rules in \( G(e_i) \) is represented by \( P(e_i) \) given in Figure 6.3.

![Diagram](image)

**Figure 6.3:** The potential execution order of commands of rules in \( G(e_i) \)

If a command \( r_j.c_j \) potentially conflicts with more than one command then the probability that the database state accessed by \( r_j.c_j \) is changed by others is calculated as follows:

As shown in Figure 6.3, there are two ordered pairs in a directed graph \( P(e_i) \) such as \([r_i.c_i,r_j.c_j] \) labelled with \( \lambda_1 \) and \([r_k.c_k,r_j.c_j] \) labelled with \( \lambda_2 \).

If the command block of \( c_i \) and the command block of \( c_k \) have not been executed yet when atomic operation \((\text{start},c_j)\) executes then the total probability \( \varepsilon \) which the
database state on which the command block of $c_j$ is executed is changed by the execution of the command blocks of $c_i$ and $c_k$ is:

$$\varepsilon = 1 - (1 - \lambda_1) * (1 - \lambda_2).$$

**Notation 6.4** For a command $c_j$ of $r_j$ $\varepsilon(r_j,c_j)$ denotes the total probability which the database state on which command $r_j,c_j$ has been executed is changed in the future by commands conflicted potentially with $r_j,c_j$. 
7.1 Data replication

Data replication is a well known technique that requires a multiversion concurrency control algorithm. In a single-version concurrency control algorithm, a read operation on a data item \( x \) reads the most recent value of \( x \). On the other hand, in a multi-version concurrency control algorithm, a read operation can read any past version of \( x \), since the multiversion technique does not destroy the old version of a data item \( x \) when a write operation is executed on \( x \).

It has been pointed out that in a multiversion algorithm, some operations which have to be delayed in a single version algorithm may be executed without being delayed. This is because, in a single version concurrency control algorithm, two operations \( p_i \) and \( p_j \) conflict if \( p_i \) reads a tuple and \( p_j \) modifies the same tuple, \( p_i \) modifies a tuple and \( p_j \) reads the same tuple, or \( p_i \) modifies a tuple and \( p_j \) also modifies the same tuple. On the other hand, in a multiversion concurrency control algorithm, only one pattern of conflict is possible such that two operations \( p_i \) and \( p_j \) conflict if \( p_i \) modifies a tuple and \( p_j \) reads the modified tuple. An operation \( p_i \) which was supposed to read a tuple is not rejected for execution by the scheduler, even though the tuple was modified already by another operation \( p_j \), because \( p_i \) can read a particular version of a tuple.

This thesis uses a multiversion concurrency control algorithm and the granularity of a data item \( x \) is considered as a tuple. If there is more than one version, then the data manager writes a data item to stable storage without destroying the old version of that data item. A tuple is produced by an insert operation. A new version of a
tuple is produced when the tuple is modified or deleted. The mapping of tuples to stable storage locations changes over time. It is therefore convenient to implement this mapping using a directory, with one entry per a tuple, giving the tuple its stable storage location. Figure 7.1 shows the mapping of a tuple in stable storage to an entry of a directory.

![Diagram of directory mapping](image)

Figure 7.1: An example of directory

**Notation 7.1** A directory of table $t_i$ is denoted by $d(t_i)$.

The directory $d(t_i)$ of table $t_i$ is implemented as an index table having a record type. When a tuple $u$ is inserted by transaction $z_i$ into $t_i$, a record $r$ consisting of the following fields is also inserted into $d(t_i)$.

- $r.tid$: a system-assigned unique tuple identifier of tuple $u$, and it represents the location where tuple $u$ is stored in a database.
- $r.zid$: the identifier of transaction $z_i$.
- $r.pty$: the priority of transaction $z_i$.
- $r.h(i)$: one dimensional array having an index variable $i$. Each one contains the identifier of a transaction which reads the tuple indicated by $r.tid$.
- $r.g(k)$: one dimensional array having an index variable $k$. Each one consists of the following fields:
7.1. Data replication

- \( r.g.tid(k) \): the identifier of a version of the tuple indicated by \( r.tid \).
- \( r.g.zid(k) \): the identifier of a transaction produced the version indicated by \( r.g.tid(k) \).
- \( r.g.pty(k) \): the priority of the transaction indicated by \( r.g.zid(k) \).
- \( r.g.h(k(j)) \): one dimensional array having an index variable \( j \). Each one contains the identifier of a transaction which reads the version indicated by \( r.g.tid(k) \).

For each record \( r \) in directory \( d(t_i) \), the one dimensional array \( r.g(k) \) keeps information about all versions of the tuple indicated by \( r.tid \). The one dimensional array \( r.g(k) \) is dynamically extended whenever a version of the tuple indicated by \( r.tid \) is produced. Thus, it is said that the version indicated by \( r.g.tid(i) \) is produced before the version indicated by \( r.g.tid(j) \) if \( i \) is less than \( j \).

The existence of multiple versions is only visible to the system not to the user transactions. Users expect the database system to behave as if there were only one version of the data item. Thus, in a multiversion concurrency control algorithm, a translation module which translates an operation on a single version data item into an operation on the specific version of the data item is needed to execute the operation. To do this, the translation module uses directories.

Given a single version operation \( (\text{get}, t_i, u) \) which operates on tuple \( u \) in table \( t_i \), the translation module translates \( (\text{get}, t_i, u) \) into \( (\text{get}, t_i, v) \), where \( v \) denotes the specific version of tuple \( u \). The translation module performs the following actions to select the specific version \( v \) of tuple \( u \):

**Implementation Model 7.1 Translation module.**

**Procedure Translation module**

*For an operation \( (\text{get}, t_i, u) \) of transaction \( z_i \),*

*begin;*

*read a record \( r \) from \( d(t_i) \), where \( r.tid \) indicates tuple \( u \).*

*\( v := u; \)*

*temp := 0;*
while not end of $r.g(k)$;
  if $(r.g.pty(k) \leq \text{the priority of } z_i \text{ and }$
    temp $\leq r.g.pty(k))$ then
    temp $:= r.g.pty(k)$;
    $v := \text{the tuple indicated by } r.g.tid(k)$;
    $k := k + 1$;
  end while;
end;

The following implementation model describes how to put information about a
version tuple produced by transaction $z_i$ into $d(t_i)$ when an operation $p_i$ issued by $z_i$
is executed on tuple $u$ in $t_i$.

**Implementation Model 7.2 Procedures:**

- *If $p_i$ insert a tuple $u$ into $t_i$, a record $r$ having information about tuple $u$ is inserted
  into directory $d(t_i)$ when tuple $u$ is inserted into table $t_i$."

  begin;
  for a directory $d(t_i)$;
  assume that the identifier of tuple $u$ is "uid";
  $r.tid := \text{"uid"}; r.zid := \text{"z_i"};$
  $r.pty := \text{the priority of } z_i;$
  insert $r$ into $d(t_i)$;
  end;

- *If $p_i$ is an update operation $(\text{update}, t_i, u)$ then the one dimensional array $r.g(k)$
of a record $r$ in directory $d(t_i)$ is directly extended when the tuple $u$ is updated."

  begin;
  $(\text{first},d(t_i))$;
while not end of d(t_i);
  (get,d(t_i),r); assume that r.tid indicates the specific version to be updated.
  r.g.tid(k + 1) := the identifier of the new version v of tuple u;
  r.g.zid(k + 1) := "zi";
  r.g.pty(k + 1) := the priority of z_i;
  write r;
end while;
end;

• If p_i is a delete operation (delete,t_i,u) then the one dimensional array r.g(k) of a record r in directory d(t_i) is directly extended when the tuple indicated by r.g.tid(k) is deleted.

begin;
  (first,d(t_i));
  while not end of d(t_i);
  (get,d(t_i),r); assume that r.tid indicates the specific version to be deleted.
  r.g.tid(k + 1) := "*"; where "*" means there is no identifier.
  r.g.zid(k + 1) := "zi";
  r.g.pty(k + 1) := the priority of z_i;
  write r;
end while;
end;

• If p_i is a get operation (get,t_i,u) then the one dimensional array r.h(i) of a record r in directory d(t_i) is directly extended.

begin;
7.1. Data replication

\[(first,d(t_i));\]
\[
\text{while not end of } d(t_i);\]
\[
\text{(get},d(t_i),r); \text{ assume that } r.tid \text{ indicates}\]
\[
\text{the specific version to be retrieved.}\]
\[
r.h(i + 1) := "z_i;\]
\[
\text{write } r;\]
\[
\text{end while;}\]
\[
\text{end;}\]

Definition 7.1 In a multi-version concurrency control algorithm, an operation \( p_i \) of \( z_i \) conflicts with an operation \( p_j \) of \( z_j \) if \( p_i \) reads a tuple which is generated by \( p_j \) where \( z_i \) and \( z_j \) are different transactions.

The following table shows a conflict relation between atomic operation, where \( (w,s(t_i),u) \) denotes an update operation, an insert operation or a delete operation.

<table>
<thead>
<tr>
<th></th>
<th>(get,s(t_i),u)</th>
<th>(w,s(t_i),u)</th>
<th>(first,d(t_i))</th>
<th>(next,d(t_i))</th>
<th>(start,c_i)</th>
<th>(end,c_i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(get,s(t_i),u)</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>(w,s(t_i),u)</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>(first,d(t_i))</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>(next,d(t_i))</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>(start,c_i)</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>(end,c_i)</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>P</td>
<td>P</td>
</tr>
</tbody>
</table>

Table 7.1 The conflict relation between specific version operations.

Assume that atomic operations in the first column are submitted by transaction \( z_j \) for execution. A "Y" entry means that for two atomic operations specified by the row and column labels, the atomic operation issued by transaction \( z_j \) conflicts with the atomic operation executed by transaction \( z_i \). A "N" entry means that for two atomic operations specified by the row and column labels, the atomic operation issued by transaction \( z_j \) does not conflict with the atomic operation executed by transaction
Process replication is achieved by clone transactions. Chapter 5 introduced the idea of process replication. For two transactions \( z_i \) and \( z_j \), if a given serial execution order is that \( z_j \) should follow \( z_i \) then a clone transaction of transaction \( z_j \) is generated when the read(\( x \)) operation of transaction \( z_j \) executes before the write(\( x \)) operation of transaction \( z_i \). In this case, the transaction \( z_j \) is called a cloning transaction and the read(\( x \)) operation of \( z_j \) is called a start-point operation. It is said that the read(\( x \)) operation of \( z_j \) potentially conflicts with the write(\( x \)) operation of \( z_i \).

To generate the clone transaction of \( z_j \) when the system executes the read(\( x \)) operation of transaction \( z_j \) the system should know that transaction \( z_i \) will submit the write(\( x \)) operation for execution in the future. This is because, if the write(\( x \)) operation of \( z_i \) is not executed in the future then it is a waste of time to generate the clone transaction of \( z_j \). If the system did not generate the clone transaction of \( z_j \) when it executed the read(\( x \)) operation of \( z_j \), then transaction \( z_j \) should be aborted when it executes the write(\( x \)) operation of \( z_i \) in the future.

It is difficult for the system to predict which operations will be submitted by transaction \( z_i \) for execution in the future if the system does not have any information about what data items transaction \( z_i \) will read and write in the future when it executes the read(\( x \)) operation of \( z_j \). To solve this problem, assume that the system has information about operations which the read(\( x \)) operation of \( z_j \) potentially conflicts with when the system executed the read(\( x \)) operation of \( z_j \). In conventional database systems, this can be done by having each transaction predeclares two sets of data items that the transaction will read and write (called, respectively, read-set and write-set).

**Definition 7.2** A clone transaction of transaction \( z_j \) is a transaction which has access to the same versions of data, it shares the same history and it is in exactly the same
stage of processing as transaction $z_j$.

It is possible for a clone transaction of $z_j$ to attempt the execution of an operation that may violates a given serialization execution order. In this case the clone transaction of $z_j$ can be cloned as well. Figure 7.2, shows that transaction $z_j$ generates its clone transaction $z_k$. The clone transaction $z_k$ of $z_j$ generates a clone transaction $z_s$. A set of clone transactions of $z_j$ is denoted by clone($z_j$) such that clone($z_j$) = \{ $z_k$, $z_s$ \}

**Notation 7.2** \| clone($z_j$) \| denotes the cardinality of clone($z_j$).

![Figure 7.2: Two clone transactions of transaction $z_j$.](image)

The strategy using the concept of clone transaction can be applied to rules in active database systems on the assumption that the execution of rules is equivalent to the execution of the respective database transactions (called rule transactions). This assumption leads to a transaction management protocol that enforces serializable execution of rule transactions such that the respective serial execution order is consistent with an order determined by the rule priorities. A directed graph $P(e_i) = (M, W)$ (Definition 6.12) is used to represent the potential execution order of commands of rules in a rooted graph $G(e_i)$(Definition 6.8).
7.2. Process replication

**Definition 7.3** Two conditions are defined as a strategy to control process replication for the respective transaction $z_j$ of any rule $r_j$ in $G(e_i)$.

1. $\|\text{clone}(z_j)\| \leq l$, and
2. $\varepsilon(r_j, c_j) \leq m$.

The parameter $l$ is used to determine the maximum number of clone transactions of $z_j$. The parameter $m$ is used to determine the maximum probability that the database state on which $r_j, c_j$ is executed is changed by other commands.

If both conditions are satisfied when the system starts executing the command block of $r_j, c_j$ then it creates a clone transaction of $z_j$. Otherwise the system delays the execution of command block of $r_j, c_j$.

Figure 7.3: The architecture of process replication

Figure 7.3 shows the architecture of process replication. Each command $c_i$ is translated into a command block containing a sequence of atomic operations. The transaction manager receives an operation $p_i$ from transaction $z_i$. If the operation $p_i$ is an atomic operation $(\text{start}, c_i)$ of $z_i$ then the transaction manager sends the operation to the clone transaction generator. Otherwise, the transaction manager sends $p_i$ to the scheduler for execution. A clone transaction of transaction $z_i$ is created when the
clone transaction generator receives an operation \( (\text{start}, c_j) \) of \( z_i \) from the transaction manager, if there is a command \( c_j \) with which the command \( c_i \) potentially conflicts and the probability of \( c_j \) to be executed in the future is lower than a given threshold. If not, the clone transaction generator sends the operation \( (\text{start}, c_j) \) of \( z_i \) to the scheduler for execution.

### 7.3 Compensation operations

This section introduces a strategy which enables a database state modified by a clone transaction to be recovered by compensation operations when the clone transaction aborts. Compensation operations are useful when a transaction consists of a sequence of independent command blocks which can be individually compensated. A compensation operation is executed when a clone transaction violates a given serial execution order while it is executing. The compensation operation uses information about whether or not each tuple was modified by a clone transaction or by a particular transaction which makes the clone transaction violate a given serial execution order. If a tuple is modified by the clone transaction or by the particular transaction then a version of the tuple is produced. If the clone transaction has read a tuple and after that, the particular transaction modifies the same tuple then a violation occurs, since the clone transaction should read the version produced by the particular transaction.

If the clone transaction aborts then its cloning transaction commences executing the same operation as the clone transaction. It is a waste of time for the cloning transaction to read all tuples which the clone transaction has read. To avoid this problem, this thesis introduces compensation operations which are executed on the version produced by the particular transaction.

Assume transactions communicate indirectly through a set of directories that has information about versions of tuples in the database such as which transactions produced which versions of tuples.

As an example, assume that \( c_i \) and \( c_j \) are any commands from a set of commands \( \{(D, t_i, \psi), (U, t_i, a_k, \psi), (R, t_i, \psi, t_j)\} \). For two transaction \( z_i \) and \( z_j \), assume that a given serial execution order is that \( z_j \) should follows \( z_i \). If \( z_j \) executes the command
block of $c_j$ before $z_i$ executes the command block of $c_i$ then the clone transaction $z_k$ of $z_j$ is created at the point when $z_j$ starts executing the command block of $c_j$. This means that transaction $z_j$ suspends its execution, on the other hand, the clone transaction $z_k$ starts executing the command block of $c_j$.

While transaction $z_i$ executes the command block of $c_i$, if a tuple that the clone transaction $z_k$ has read is modified by transaction $z_i$ then the clone transaction $z_k$ should be aborted since it violates a given serial execution order.

If $z_k$ and $z_i$ operate on the same tuple $u$ and $z_k$ executed before $z_i$ then one of the following four cases occurs:

1. The clone transaction $z_k$ reads tuple $u$ and produces a version $v$ of $u$. The transaction $z_i$ reads the same tuple $u$ as $z_k$ and produces a version $w$ of $u$.

2. The clone transaction $z_k$ only reads tuple $u$. The transaction $z_i$ reads the same tuple $u$ as $z_k$ and produces a version $v$ of $u$.

3. The clone transaction $z_k$ reads tuple $u$ and produces a version $v$ of $u$. The transaction $z_i$ only reads the same tuple $u$ as $z_k$.

4. The clone transaction $z_k$ only reads tuple $u$. The transaction $z_i$ also only reads the same tuple $u$ as $z_k$.

If clone transaction $z_k$ aborts then its cloning transaction $z_j$ starts executing the same operations executed by $z_k$. Table 7.2 shows the versions generated by two transactions $z_k$ and $z_i$. Assume that an atomic operation $p_j$ is in the command block of $c_j$ and an atomic operation $p_i$ is in the command block of $c_i$.

<table>
<thead>
<tr>
<th>Row</th>
<th>Tuple</th>
<th>$p_j, z_k$</th>
<th>$p_i, z_i$</th>
<th>$p_j, z_j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$u$</td>
<td>version $v$</td>
<td>version $w$</td>
<td>read $w$</td>
</tr>
<tr>
<td>2</td>
<td>$u$</td>
<td>*</td>
<td>version $w$</td>
<td>read $w$</td>
</tr>
<tr>
<td>3</td>
<td>$u$</td>
<td>version $v$</td>
<td>*</td>
<td>read $u$</td>
</tr>
<tr>
<td>4</td>
<td>$u$</td>
<td>*</td>
<td>*</td>
<td>read $u$</td>
</tr>
</tbody>
</table>

Table 7.2 The versions generated by operations.
7.3. Compensation operations

- Row 1 shows that cloning transaction $z_j$ reads the version $w$ of tuple $u$ since $w$ is the most recent version of tuple $u$ except the version produced by its clone transaction $z_k$.

- Row 2 shows that cloning transaction $z_j$ is executed on the version $w$ of tuple $u$, since version $w$ is the most recent version of tuple $u$.

- Row 3 shows that cloning transaction $z_j$ is executed on tuple $u$, since tuple $u$ is the most recent version of tuple $u$ except the version produced by its clone transaction $z_k$.

- Row 4 shows that cloning transaction $z_j$ is executed on tuple $u$, since tuple $u$ is the most recent version of tuple $u$.

As a result, it is not necessary for cloning transaction $z_j$ to read tuples which have not been modified by transaction $z_i$. Thus the compensation operation of the command block of $c_j$ is needed to execute on tuples modified by $z_i$.

The following implementation model shows actions to be performed by compensation operations when the clone transaction $z_k$ aborts.

**Implementation Model 7.3 Procedures:**

- In the case where $p_i$ is an insert, an update or a delete operation and $p_j$ is an update operation, a delete operation, or a retrieval operation.

  begin;
  for a directory $d(t_i)$;
  (first, $d(t_i)$);
  while not end of $d(t_i)$;
  (get, $d(t_i)$, $r$);
  assume that a record $r$ is derived and $r.tid$ indicates tuple $u$.
  search for the most recent version $w$ produced by $z_i$.
  if found $w$ then test whether or not $z_k$ has executed on $w$;
  if $z_k$ executed on $w$ then exit;
else execute $p_j$ on the version $w$ and exit;

else test whether or not $z_k$ executed on tuple $u$;

if $z_k$ executed on $u$ then exit;

else execute $p_j$ on $u$ and exit;

$(next,d(t_i))$;

end while;

end;

• In the case where $p_i$ is an insert, an update or a delete operation and $p_j$ is an insert operation then the compensation does not occur.

• In the case where $p_i$ is a retrieval operation and $p_j$ is also a retrieval operation then the compensation does not occur.

As a result, this compensation model produces the same results as the execution clone transaction $z_i$.

7.4 Extended serialization graph

In active database systems, rules are triggered by mapping events to atomic operations. An event $e_i$ associated with command $c_i$ is created when the operation $(end,c_i)$ in the command block of $c_i$ is executed successfully. The event $e_i$ may trigger rules and the triggered rules may generate new events that may trigger other rules and so on, conceivably resulting in an arbitrarily deep hierarchy of triggered rules. A rooted graph $G(e_i) = (N,V)$ resulting from static rule analysis represents the hierarchy of triggered rules. The potential execution order of commands of rules in $G(e_i)$ is represented by a directed graph $P(e_i) = (M,W)$.

If an event $e_i$ is generated by an external transaction then the corresponding rooted graph $G(e_i)$ and the directed graph $P(e_i)$ are selected by the rule manager to trigger rules and to know a given serial execution order assigned to commands of rules in $G(e_i)$. All rules in $G(e_i)$ are executed as part of the external transaction which generated
the root node of $G(e_i)$. The external transaction can be created either by database application programs, or a triggered rule where the coupling mode is separate.

The concurrent execution of external transactions is represented by the extended serialization graph ESG such that $ESG = (N,E_s,E_d)$ where $N$ is a set of identifiers of non-committed external transactions, called nodes, $E_s$ is a set of 'solid' edges such that $E_s \subseteq N \times N$, and $E_d$ is a set of 'dashed' edges such that $E_d \subseteq N \times N$.

**Definition 7.4** Given an extended serialization graph $ESG = (N,E_s,E_d)$, the dependency between nodes is defined as follows:

- If an edge $[z_i,z_j] \in E_s$, then it is said that external transaction $z_j$ directly depends on external transaction $z_i$.

- The set of external transactions that directly depends on external transaction $z_i$ is denoted by $d(z_i)$.

- The set of external transactions that depends on external transaction $z_i$ is denoted by $d^*(z_i)$, and recursively defined as follows:
  
  $d^*(z_i) = \{z_j | z_j \in d(z_i) \text{ or there exists } z_k \text{ such that } z_k \in d^*(z_i) \text{ and } z_j \in d(z_k)\}$.

- $|d^*(z_i)|$ denotes a cardinality of the set $d^*(z_i)$. 

The present chapter proposes two hybrid algorithms for concurrent execution of transactions in active database systems:

1. Hybrid concurrency control algorithms for external transactions in active database systems.

2. Hybrid concurrency control algorithms for the respective transactions of rules in G(e_i).

The two hybrid algorithms have the ability to dynamically adjust the behaviour of concurrency control algorithms to changing environmental parameters.

- A H(n,k) concurrency control algorithm for external transactions is defined as a strategy, where for any external transaction z_i

  1. || d*(z_i) || ≤ n.
  2. || p(z_i) || ≤ k.

The parameter n of H(n,k) determines the acceptable size of cascading abort of external transactions and the parameter k determines the total number of external transactions that execute concurrently with external transaction z_i.

- A H(l,m) concurrency control algorithm for rule transactions is defined as follows. For the respective transaction z_j of any rule r_j in G(e_i)

  1. || clone(z_j) || ≤ l.
  2. || e(r_j,c_j) || ≤ m.
The parameter \( l \) of \( H(l,m) \) determines the maximum number of clone transactions of rule transaction \( z_j \) and the parameter \( m \) determines the maximum probability that the database state on which the command \( r_j.c_j \) is executed is changed in the future by other commands potentially conflicted with \( r_j.c_j \).

### 8.1 The properties of \( H(n,k) \) and \( H(l,m) \) algorithms

The classification of concurrency control algorithms depends on the kinds of action which a scheduler undertakes when it receives an operation that introduces a risk of violation of serializable execution. In a locking algorithm, the scheduler delays the execution of an operation if the operation attempts to access a data item that has been accessed earlier by another operation in a conflict mode. In a non-locking algorithm, the scheduler executes all operations immediately but it has a responsibility to test whether or not the immediate execution causes a non-serializable execution.

In both \( H(n,k) \) and \( H(l,m) \) algorithms, when the scheduler receives an operation in a conflict mode, the scheduler has a responsibility to decide whether it immediately executes the operation or delays the execution of the operation.

- **H\((n,k)\) algorithm**

  As far as the concurrency control of external transactions is concerned, the \( H(n,k) \) algorithm reduces itself either to a strict two-phase locking algorithm or a serialization graph testing, since the parameters \( n \) and \( k \) are used to control the behaviour of concurrent execution of external transactions.

  As shown in section 4.6.3, if \( n = 0 \) and \( k = 0 \), the serialization graph analyzer returns a delay to the scheduler whenever it receives a pair \([z_i,z_j]\) from the conflict analyzer, because the related hybrid conditions are not always satisfied. Therefore an external transaction should be suspended if its operation conflicts with an operation executed already by another external transaction. This avoids the cascading abort phenomenon and also detects a cycle which represents a dead lock phenomenon from the locking algorithm's point of view because the system maintains an extended serialization graph including dashed edges representing a
wait-for relationship.

If \( n \to \infty \) and \( k \to \infty \), the extended serialization graph is the same as the serialization graph, since there are no dashed edges in the extended serialization graph. Therefore the serialization graph analyzer always returns an execution message to the scheduler whenever it receives a pair \([z_i, z_j]\) from the conflict analyzer, where \( z_i \) and \( z_j \) are external transactions.

- **H\((l,m)\) algorithm**
  
  As far as the concurrency control of respective transactions of rules in \( G(e_i) \) is concerned, the system using \( H(l,m) \) algorithm has the responsibility of deciding whether or not to execute immediately an operation issued by the respective transactions of rules in the hierarchy of an external transaction.

  The decision depends on two conditions: one is related to parameter \( l \), the other is related to parameter \( m \). For instance, assume that two rules \( r_i \) and \( r_j \) are triggered by the same event, where \( r_i \) has a condition in its condition-part such that it is always true and \( r_i \) has a command \( c_i \) in its action-part which inserts a tuple into table \( t_i \). The \( r_j \) has a retrieval command \( c_j \) in its action-part. Assume that a given execution order is that \( r_j \) should follow \( r_i \).

  When the system executes the retrieval command of \( r_j \) on table \( t_i \) before \( r_i \) it knows that the insert command of \( r_i \) will be executed at any time in the future, since the condition in the condition-part of \( r_i \) is always true, i.e., the probability of the insert operation to be executed in the future is equal to 1.

  By the condition, \( \epsilon(r_j.c_j) \leq m \), the system should put the retrieval command of \( r_j \) into its delay queue, if the parameter \( m \) has a lower value than 1. It is necessary to delay the execution of the retrieval command since its execution is able to violate a given serial execution order in the future.

  By the condition, \( ||\text{clone}(z_j)|| \leq l \), where \( z_j \) is the respective transaction of \( r_j \), if \( z_j \) has any number of clone transactions less than \( l \), then it has the possibility of having an additional clone transaction. It is necessary to restrict the number of clone transactions of \( z_j \) because of the cascading abort of clone transactions.
8.1. The properties of H(n,k) and H(l,m) algorithms

If \( r_j . c_j \) satisfies both conditions then the system executes the retrieval command, otherwise the system puts the retrieval command into its delay queue.

The hybrid approach has an ability to dynamically adjust the behaviour of concurrency control algorithms as it changes the given thresholds of environmental parameters. It is proved that for extreme values of parameters, this strategy reduces itself either to altruistic locking approach or to serialization graph testing.

In section 9.3, if \( l = 0 \) and \( m = 1 \), then the clone transaction manager returns a delay to the scheduler whenever it receives a command which potentially conflicts with another command to be executed in the future. This means that for a deterministic execution paradigm, a command \( c_j \) which potentially conflicts with another command \( c_i \) to be executed in the future should be delayed until \( c_i \) executes if a given serial execution order is that \( c_j \) should follow \( c_i \). It is recognized that \( c_i \) has a potential lock even though \( c_i \) is not executed yet. Therefore, if the execution order of rules is deterministic, the hybrid algorithm, \( H(l = 0, m = 1) \) can be thought of as a locking algorithm as far as the concurrent execution of respective transactions of rules is concerned.

If \( l \rightarrow \infty \) and \( m = 0 \), then the clone transaction manager always generates a respective clone transaction to a command \( c_j \) whenever it receives \( c_j \) from the transaction manager, where \( c_j \) potentially conflicts with another command \( c_i \) to be executed in the future. It is recognized that the hybrid algorithm \( H(l \rightarrow \infty, m = 0) \) can be thought of as a serialization graph testing, since it does not delay the execution of any command issued by respective transactions of rules triggered by the same event.

Consequently, as far as the concurrency control of respective transactions of rules is concerned, the \( H(l,m) \) algorithm reduces itself either to altruistic locking approach or to serialization graph testing. It depends on the given thresholds of the environmental parameters.
8.2 Proof of correctness of $H(n,k)$ algorithm

In general, a database system is required to provide the mechanism for executing transactions such that each transaction preserves the integrity constraints of the database; if executed alone in a consistent state, it leaves the database in a consistent state. The simplest approach to ensure this is to execute the transactions serially: each transaction executes from beginning to end without interleaving of the operations of different transactions. However this approach is too restrictive, since if a pair of transactions do not share a common data item, it is clear that interleaving their operations cannot violate any integrity constraints. Much research has been devoted to increasing the efficiency of databases by allowing several transactions to execute concurrently. Correctness criteria have been developed to ensure that a concurrent execution of a set of transactions is equivalent to some serial execution of the same set of transactions. This notion of equivalence to a serial execution is referred to as serializability, and is accepted as the cornerstone of database correctness.

This section describes the hybrid approach in the concept of serializability to prove the correctness of the hybrid approach. The hybrid approach is based on the assumption that the order of a serial execution of respective transactions of rules triggered by the same event is determined as priorities of rules given by a rule designer. For convenience, the respective transactions of rules triggered by an event which is generated from the execution of an operation issued by an external transaction are called sibling transactions and all transactions in the hierarchy of an external transaction are called descendents of the external transaction.

**Definition 1** A history generated by the hybrid algorithm is serial if for any sibling transactions $z_i$ and $z_j$ in the history, all operations of $z_i$ and all operations of descendents of $z_i$ complete its execution, before any operations of $z_j$ and any operation of descendents of $z_j$ execute, where the priority of $z_i$ is less than $z_j$.

**Definition 1** A history, denoted by $h_i$, generated by the hybrid algorithm is serializable if it is equivalent to a serial history $h_j$. Two histories $h_i$ and $h_k$ are equivalent to each other if the two histories have the same read-from relationship.
Theorem 1  An extended serialization graph ESG representing the execution order of external transactions over a history $h_i$ is serializable if and only if there are no cycles in ESG.
Chapter 9

Scheduling Algorithm for Rules in $G(e_i)$

The present section introduces a scheduling algorithm for rules in $G(e_i)$. It integrates data replications, process replications, and compensation operations into a typical model of an active database system. The scheduling algorithm is based on an assumption that execution of rules in $G(e_i)$ is equivalent to the execution of respective database transactions. The assumption leads the scheduling algorithm to a transaction management protocol that enforces serializable execution of database transactions such that the respective serial order is consistent with an order determined by the rule priorities.

Assume that each command $c_i$ in rules is translated into a command block containing a sequence of atomic operations, where an atomic operation $(\text{start}, c_i)$ is used to begin executing the block and an atomic operation $(\text{end}, c_i)$ is used to end the execution of the block. The potential execution order of commands of rules in $G(e_i)$ is represented by a directed graph $P(e_i) = (M,W)$ resulting from static analysis for rules in the active database system.

As shown in Figure 9.1, the typical model of an active database system consists of six modules: Transaction manager, Rule manager, Clone transaction manager, Scheduler, Conflict analyzer, and Data manager. The six modules send requests to, and receive replies from, the modules directed by arrows.
9.1 Scheduling algorithms

The following algorithms describe in detail the actions to be performed by the modules.

Algorithm 9.1 Transaction Manager

- **Input:** The transaction manager receives one of the following inputs.
  - Operation $p_i$ submitted by transactions $z_i$ or
  - A clone message (clone,$p_i$,z$_i$) or
  - A delay message (delay,$p_i$,z$_i$).

- **Output:** The transaction manager performs the following related actions depending on input.
  - The operation $p_i$ of transaction $z_i$ or
  - An event message (event,$e_i$,z$_i$)

- **Method:**
  - The case that the transaction manager receives an operation $p_i$ from transaction $z_i$:
    * if $p_i$ is the start-block operation of a command $c_i$ such as (start,$c_i$), then the transaction manager sends $p_i$ to the clone transaction manager.
9.1. Scheduling algorithms

* Otherwise, the transaction manager sends \( p_i \) to the scheduler.

- The case that the transaction manager receives a clone message \((\text{clone}, p_i, z_i)\) from the clone transaction manager:
  * it adds a pair \([z_c, z_i]\) to the set \( G \) of pairs representing the relationship between clone transaction \( z_c \) and its cloning transaction \( z_i \), and
  * it puts \( p_i \) of cloning transaction \( z_i \) to a delay queue, and
  * it sends \( p_i \) issued by clone transaction \( z_c \) to the scheduler.

- The case that the transaction manager receives a delay message \((\text{delay}, p_i, z_i)\) from the clone transaction manager:
  * it puts \( p_i \) issued by transaction \( z_i \) into a delay queue.

- The case that the transaction manager receives a reply on \( p_i \) from the scheduler:
  * if \( p_i \) is the end-block operation of a command \( c_i \) such as \((\text{end}, c_i)\), then the transaction manager creates an event message \((\text{event}, c_i, z_i)\) and sends it to the rule manager.
  * Otherwise the transaction manager receives the next operation of \( p_i \) from transaction \( z_i \).

Algorithm 9.2 Clone Transaction Manager

- Input:
  - An operation \( p_i \) submitted by the transaction manager and
  - A directed graph \( P(e_i) = (M, W) \) resulting from the static rule analysis.

- Output:
  - A clone transaction \( z_c \) of \( z_i \) with a clone message \((\text{clone}, p_i, z_i)\) or
  - A delay message \((\text{delay}, p_i, z_i)\).

- Method: If the clone transaction manager receives an operation \( p_i \) of \( z_i \) from the transaction manager:
- The clone transaction manager finds all pairs \((r_j, c_j, r_i, c_i)\), where the \(r_i\) of \(r_i, c_i\) is the corresponding rule of transaction \(z_i\) and the command \(c_i\) of \(r_i, c_i\) potentially conflicts with the command \(c_j\) of \(r_j, c_j\), but the start-operation of command \(c_j\) is not executed yet by the respective transaction \(z_j\).

- For each pair \((r_j, c_j, r_i, c_i)\) found, the clone transaction manager takes the probability, denoted by \(\lambda(r_j)\), that the command block of \(c_j\) of \(r_j\) is executed by the corresponding transaction \(z_j\) in the future.

- For all \(\lambda(r_j), \ldots, \lambda(r_k)\), the transaction manager calculates the following total probability \(\epsilon\) that transaction \(z_i\) aborts due to the violation of a given execution order.

The total probability \(\epsilon = 1 - ((1 - \lambda(r_j)) \cdots (1 - \lambda(r_k)))\).

* If \(\epsilon \geq\) a given threshold then the clone transaction manager creates a message \((\text{delay}, p_i, z_i)\) and sends it to the transaction manager.

* Otherwise, the clone transaction manager creates a clone transaction \(z_c\) of \(z_i\) and a clone message \((\text{clone}, p_i, z_i)\). The clone transaction manager sends the clone message to the transaction manager.

**Algorithm 9.3 Rule Manager**

- **Input:**
  - An event message \((\text{event}, z_i, e_i)\) submitted by the transaction manager.
  - A set of rooted tree \(G = (N, V)\) resulting from static rule analysis.

- **Output:**
  - The respective transactions of rules triggered by the event \(e_i\) of the message \((\text{event}, z_i, e_i)\).

- **Method:** When the rule manager receives a message \((\text{event}, e_i, z_i)\) from the transaction manager it triggers rules related to event \(e_i\) and creates the respective transactions of rules and sends the transactions to the transaction manager for execution.
Algorithm 9.4 Scheduler

- **Input:** The scheduler receives one of the following inputs:
  - An operation $p_i$ of $z_i$ from the transaction manager or
  - A reject message or an execute message from the conflict analyzer:

- **Output:**
  - An operation $p_i$ of $z_i$.

- **Method:**
  - The case that the scheduler receives one of the following messages from the conflict analyzer (Algorithm 9.5):
    - An execute message $(execute, p_i, z_i)$: The scheduler appends $p_i$ to execution history and sends $p_i$ to the data manager.
    - A reject message $(reject, p_i, z_i)$: The scheduler submits abort actions for transaction $z_i$ into its input queue.
  - The case that the scheduler receives an operation $p_i$ from the transaction manager:
    - if $p_i \notin \{\text{commit, abort}\}$ then the scheduler sends $p_i$ to the conflict analyzer. The conflict analyzer returns the corresponding messages to the scheduler.
    - if $p_i$ is a commit operation then the scheduler performs the following actions: if there exist a non-committed transaction $z_k$ such that $z_i$ reads a tuple generated by transaction $z_k$ then the scheduler inserts $p_i$ into its delay queue. Otherwise the scheduler executes all commit actions for transaction $z_i$.
    - if $p_i$ is an abort operation then the scheduler performs the following actions: if $z_i$ is a clone transaction then the scheduler extracts $p_i$ issued by the cloning transaction of $z_i$ from the delay queue and puts the compensation operation of $p_i$ into an input queue. The scheduler executes
all abort actions for $z_i$ when it receives a reply on the compensation operation of $p_i$. Otherwise, the scheduler executes all abort actions for transaction $z_i$.

Algorithm 9.5 Conflict Analyzer

- **Input:**
  - An operation $p_i$ of $z_i$ from the scheduler and
  - Directory $d(t_i)$ if $p_i$ operates on table $t_i$.
- **Output:** A message either $(\text{execute}, p_i, z_i)$ or $(\text{reject}, p_i, z_i)$
- **Method:**
  - If operation $p_i$ of $z_i$ operates on table $t_i$ then the conflict analyzer extracts directory $d(t_i)$ from a set of directories
  - The case that $p_i$ is a get operation on tuple $u$ in $t_i$,
    * The conflict analyzer searches $d(t_i)$ for the respective record $r$ of tuple $u$.
    * The conflict analyzer finds the most recent version $v$ of tuple $u$, where $v$ is produced by a transaction $z_j$ of which the priority is less than or equal to $z_i$.
    * If there is a transaction $z_k$ which has read the most recent version $v$ and the priority of $z_k$ is larger than $z_i$ then the conflict analyzer returns both messages, $(\text{reject}, p_k, z_k)$ and $(\text{execute}, p_i, z_i)$, to the scheduler. Otherwise, the conflict analyzer returns a message $(\text{execute}, p_i, z_i)$ to the scheduler.

Algorithm 9.6 Data Manager

- **Input:**
  - An operation $p_i$ issued by transaction $z_i$ and
  - A directory $d(t_i)$ if $p_i$ is operates on the most recent version $v$ of $u$ in table $t_i$ and
9.2. Termination algorithm

- A table $t_i$.

- Output:
  - A modified table $t_i$ and
  - A modified directory $d(t_i)$

- Method:
  - If $p_i$ operates on a version $v$ of tuple $u$ then the operation has information about a record $r$ in $d(t_i)$.
  - The data manager either reads tuple $v$ or modifies tuple $v$.
  - If tuple $v$ is modified then the data manager creates a new version of tuple $v$ and puts all information related to the new version into the corresponding fields of record $r$ in $d(t_i)$.
  - If tuple $v$ is read then the data manager puts information related to $p_i$ into the fields of record $r$ in $d(t_i)$.

9.2 Termination algorithm

In an active database system, if one rule's action triggers another rule (or even itself again), and this rule’s action triggers a third one, the result of this might be an infinite cyclic execution (i.e., a termination problem) of some rules in the system. The infinite cyclic execution is detected by the rule manager when a rule appears more than once in a unique path from the rule to the root node of $G(e_i)$.

There are two types of infinite cyclic executions. One produces an infinite number of possible database states, the other produces an infinite repetition of a database state. The former could be prevented if a rule designer gives some restrictions to a command which causes the infinite cyclic execution. The latter is detected when a command which causes the infinite cyclic execution is executed on a database state such that it must be repeated more than once.

This thesis tests the latter type in the following way. From the execution history produced by the scheduler, execution symbols can be generated. Let $(s(DB),r_i.c_i)$ be
an execution symbol meaning that the command block of \( c_i \) of rule \( r_i \) is executed on a
database state \( s(DB) \). A cyclic execution history (denoted by \( H \)) can be represented by
a sequence of execution symbols such that \( H = (s(DB), r_i . c_i), \ldots, (s(DB'), r_i . c_i) \).

If the most recent versions of data items in \( s(DB') \) have the same values as in \( s(DB) \) then it is said that the history \( H \) has an infinite repetition cyclic execution order.

Assume that the most recent version of a tuple \( t \) is denoted by the letter \( s \) possibly
followed by a subscript which identifies this particular version. The history \( H \) has an
infinite repetition cyclic execution order if \( s(DB') \) has tuples satisfying the following
conditions:

- If \( s_i \in s(DB) \) is deleted by an execution symbol, then the most recent version \( s_k \)
of tuple \( t \) should be in \( s(DB') \), where \( s_k = s_i \).
- If a tuple \( t \notin s(DB) \) is inserted by an execution symbol and the tuple \( t \) is updated
  later, then the version of \( t \) should be deleted from \( s(DB') \).
- If \( s_i \in s(DB) \) is updated by an execution symbol, then the most recent version
  \( s_k \) of tuple \( t \) should be in \( s(DB') \), where \( s_i = s_k \).
- If \( s_i \in s(DB) \) is updated and after that deleted, then \( s(DB') \) should have a tuple
  which has the same value as \( s_i \).

For convenience, assume that all execution symbols in \( H \) are executed on table \( t_k \).
Assume that each execution symbol in \( H \) associates with a \( \Delta \) table which contains tuples
resulting from the execution of the respective command block of the execution sym-
bol. If \( c_i \) in an execution symbol \( (s(DB), r_i . c_i) \) is an insert or a delete command, then
\( \Delta_i(I,DB) \) or \( \Delta_i(D,DB) \) which contains tuples inserted or deleted by the respective com-
mand block is produced. If \( c_i \) in an execution symbol \( (s(DB), r_i . c_i) \) is an update com-
mand, and a tuple \( t \) is updated to \( s_k \) then \( \Delta_i(U.old,DB) \) and \( \Delta_i(U.new,DB) \) are pro-
duced, where \( \Delta_i(U.old,DB) \) contains the old version \( t \) of \( s_i \) updated and \( \Delta_i(U.new,DB) \)
contains the tuple \( s_i \) that is a new version of \( t \).

A one-dimensional table \( F \) is constructed to compare two database states. An entry
in \( F \) consists of the same attributes of table \( t_k \) and an additional attribute denoted by
op-type. If table \( F \) has no record after the execution of the following procedure, the
9.2. Termination algorithm

cyclic execution becomes an infinite repetition cyclic execution.

**Procedure of termination algorithm**

Input: $\Delta_i(I,t_k), \Delta_k(D,t_k)$ and $\Delta(U.new,t_k)$

Output: Table F.

begin

accept a $\Delta$ table: assume that each $\Delta$ table
is accepted by the same order as the respective execution symbol occurs in H.

For each tuple $s_k \in \Delta_i(I,t_k)$,

search F for a record r having the same key value as $s_k$

if found and

if r.op-type = I and r = $s_k$ then delete r from F

else exit

else

insert the corresponding record s of $s_k$ into F,

set s.op-type := D.

For each tuple $s_k \in \Delta_i(U,t_k)$

search F for a record r having the same key value as $s_k$

if found and

if r.op-type = U and

if r = $s_k$ then delete r from F

else exit

else exit

else

insert a record s which corresponds to
the old tuple of $s_k$ into F,

set s.op-type := U

For each tuple $s_k \in \Delta_k(D,t_k)$,

search F for a record r having the same key value as $s_k$

if found and
9.2. Termination algorithm

if r.op-type = D then delete r from F
else
  if r.op-type = U then set r.op-type := I
  else exit
else
  insert the corresponding record s which corresponds to $s_k$ into F,
  set s.op-type := I.
Chapter 10

Conclusions

In active database systems, rule processing occurs when a database state is changed by the execution of operations requested by users. Such operations cause events and each event triggers rules. The execution of rules can trigger additional rules or can trigger the same rules. The unpredictable and non-deterministic behaviour of rule processing leads to an unexpected result such as unpredictable final database states or infinite cyclic executions of rules.

This thesis has introduced a strategy which solves the problem of non-deterministic execution of rules (Confluence problem) and the problem of cyclic execution (Termination problem) in active database systems.

The strategy is based on the assumption that the execution of rules in active database systems has the same semantics as the execution of transactions in databases systems. The strategy enforces the concurrent execution of the respective transactions of rules. Histories representing the concurrent execution of the respective transactions of rules are serializable histories in which the execution order of transactions is the same as a serial execution order given by a rule designer.

To find a more efficient solution for the concurrent execution of transactions, this thesis has used the concept of data replication and proposed a new transaction model, i.e., clone transactions.

The benefit of multiple versions for concurrency control is to reduce the transaction rejection and thus to increase the degree of concurrency, because only one pattern of conflict, i.e., read-from pattern is possible in multiversion concurrency control algorithms. An obvious cost of maintaining versions is storage space. But maintaining multiple versions may not add much to the cost of concurrency control, because the
versions may be needed anyway by the recovery algorithm.

In multiversion concurrency control algorithms, the data manager may write a data item to stable storage without destroying the older version of that data item. The mapping of data items to stable storage locations changes over time. It is therefore convenient to implement this mapping using a directory, with one entry per data item, giving the name of the data item and its stable storage location. This thesis introduced a directory per a table and each entry of the directory contains information about a tuple and its versions. Such a directory defines the state of the table.

The existence of multiple versions is only visible to the scheduler and the data manager, not to user transactions. To process operations from transactions, the scheduler must translate operations on single version data items into operations on specific versions of those data items. The directory of a table is used to not only map tuples of the table to stable storage locations but also translate operations on single version data items into specific versions of those data items.

This thesis introduced the concept of process replication. The process replication is achieved by transaction cloning. For two transactions $z_i$ and $z_j$, where a given serial execution order is that $z_j$ should follow $z_i$, a clone transaction of transaction $z_j$ is generated if the operation $p_j$ of transaction $z_j$ executes before the operation $p_i$ of transaction $z_i$, where $p_j$ potentially conflicts with $p_i$, and the probability of $p_i$ to be executed in the future is lower than a given threshold. In active database systems, it is possible to assign probabilities to the commands of rules, since all rules have predefined commands.

A clone transaction of transaction $z_j$ is a transaction which has access to the same versions of data, it shares the same history and it is in exactly the same stage of processing as transaction $z_j$. As a clone transaction plays the role of a reconnaissance transaction that performs the actions in advance of its cloning transaction, it reduces the possibility that the cloning transaction re-executes entire operations which have been executed already.

Moreover this thesis introduced a concept of compensation operations which recover a database state affected by a transaction that has violated a given serial execution order. The compensation operations have a function which selects tuples needed for
the recovery from the affected database state. As mentioned earlier, the thesis used directories to translate operations on a single version of data items into specific versions of those data items. As directories have information about tuples and their versions, the compensation operations execute on the selected tuples without repetition of the entire database. This is useful if a transaction consists of independent actions that can be individually compensated.

The classification of concurrency control algorithms depends on the kind of action a scheduler undertakes when it receives an operation that introduces a risk of violation of serializable execution. One possibility is that the scheduler may delay execution of an operation until the risk of non-serializable execution disappears. Such behaviour is typical of locking algorithms. The other possibility is to risk non-serializable execution and to schedule the operation for immediate execution. This approach is typical of non-locking strategies.

This thesis has introduced a new family of concurrency control algorithms that use both locking and non-locking strategies to achieve better performance of concurrently running database transactions. An important advantage of hybrid concurrency control algorithms proposed in this thesis, is their ability to dynamically adjust their behaviour to the changing parameters of the environment in which they are working. As far as the concurrency control of external transactions is concerned, the $H(n,k)$ algorithm reduces itself either to a strict two-phase locking algorithm or a serialization graph testing, since the parameters $n$ and $k$ are used to control the behaviour of concurrent execution of external transactions. As far as the concurrency control of respective transactions of rules in $G(e_i)$ is concerned, the $H(l,m)$ algorithm reduces itself either to altruistic locking algorithm or serialization graph testing, since the parameters $l$ and $m$ are used to control the behaviour of concurrent execution of rule transactions.

The confluence problem does not occur, since all rules are executed following the priorities of rules, and the priorities are used to define the partial execution order of rules. Although the partial execution order of rules represents the serial execution order, this thesis has introduced a data replication, process replication, compensation operations and hybrid concurrency control approach to enhance the concurrency of
transactions. The termination problem has been solved by an algorithm which compares two database states when an infinite cyclic execution occurs.

This thesis discussed only theoretical results. Further work is needed to study the applicability of hybrid algorithms and analysis of their performance in the real world environment.
Bibliography


*Proceedings of the 13th VLDB Conference*, Brighton, 1897.


ceedings of IEEE Int. Conference on Intelligent and Cooperative In­

[Etz94] O. Etzion, "An Alternative Paradigm for Active Databases", Proceed­
ings of 4th IEEE Int. Workshop on Research issues in Data Engineering  
and Active Database Systems, Houston, TX, USA, Feb. 1994.

systems", Proc. First International Conf. Rules in Database Systems,  

[GJ91] N. H. Gehani and H. V. Jagadish, "Ode as an Active Database: Con­
straints and Triggers", Proc. 17th International Conf. Very Large Data­

IEEE Bulletin of the TC on Data Engineering, 15(1-4), pages 19–22,  

Proc. of the 7th Int. Conf. on very Large Data Base, pages 144–154,  


[Han92] E. N. Hanson, "Rule Condition Testing and Action Execution in  


BIBLIOGRAPHY


