The adaptive serializable snapshot isolation protocol for managing database transactions

Yang Yang
University of Wollongong

The Adaptive Serializable Snapshot Isolation Protocol for Managing Database Transactions

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

Master of Computer Science by Research

from

UNIVERSITY OF WOLLONGONG

by

Yang Yang

Computer Science Department
February 2007

Dedicated to

My parents

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.

Yang Yang
February 15, 2007

Abstract

In this thesis, concept of database concurrency control, computational models of database transaction, the correct criterias of concurrent execution of transactions and concurrency control algorithms such as two phase locking, serialization graph testing, Snapshot Isolation are reviewed. A graph based mechanism is proposed for preserving Snapshot Isolation protocol(SI) serializable at run-time. Firstly, we present Dynamic Managed Snapshot Isolation Serialization Graph(called DSISG). By using this mechanism, non-serializable transactions under Snapshot Isolation protocol can be detected at run-time. Secondly, in order to guarantee the effectivity of DSISG, a new model of database transaction(segmented transaction model) is proposed. Thirdly, an algorithm of managing a hierarchical structured acyclic graph is presented. The run-time characterzing of non-serializable transaction under Snapshot Isolation protocol will be more efficient when this hierachical graph structure is applied to DSISG. We also summarize the contributions of this thesis and formulate some open problems.

Acknowledgments

I would like to extend my sincere thanks to my supervisor Dr. Janusz R. Getta without whose invaluable assistance this thesis would not have been possible.

My thanks also go to the technical staff in the school of Information Technology and Computer Science for the help they gave me.

I am also grateful to my parents and friends for their supports throughout this work.

Contents

List of Figures

Chapter 1

Introduction

In the modern database systems, it is important that transactions submitted by different users can be handled simultaneously. Database transaction is an execution of a program submitted by a user accessing a shared database. In order to preserve the integrity and consistency of a database, the concurrent execution of transactions requires to be equivalent to a serial execution over the same set of transactions. Concurrency control is the mechanism that guarantees the serializability of concurrent execution.

Database concurrency control algorithms can be classified into two categories: Pessimistic Protocol and Optimistic Protocol. Algorithms that belong to Pessimistic Protocol are based on locking mechanism. The simultaneous accesses on shared data items are managed by locks, which can be set on and removed from data items on behalf of transactions. When a data item is locked by a transaction, other transactions that want to access this data item will be suspended until the lock on it is released. On the other hand, algorithms that belong to Optimistic Protocol are based on validation after the execution. Operations on shared data items are alway allowed to be executed. Then the system will verify the serializability of schedule frequently. Transactions which may harm the serializability of schedule will be aborted.

Snapshot Isolation protocol(SI) is an optimistic concurrency control algorithm that has been widely implemented by database system vendors. It precludes many concurrency problems by managing multiple versions of data items. However SI can not guarantee the serializability in all cases. During the concurrent execution of transactions, the violation of constrain which involves numbers of data items can not be prevented by SI. Previously, researcher proposed a mechanism that can characterize the serializability of Snapshot Isolation protocol. The general idea of this mechanism is to characterize the non-serializable transactions by evaluating an ad hoc Interference

Graph of all the transactions that can be executed concurrently. However, this mechanism can only be implemented in *system design pahse*. The aim of this thesis is to provide an efficient mechanism that makes Snapshot Isolation protocol serializable at *run time*. The problem is solved through the following steps.

Firstly, in order to decrease the time spent on characterizing the serializability, I propose another mechanism which can reach the same objective. The acyclicity of a graph MVSG is used to characterize the serializability of schedule running under SI. Because the cost of aborting a transaction at the very end will become unacceptable when transaction has been running for a long period, I provide a dynamic managed variation of MVSG(called DSISG) so that non-serializable transactions can be detected earlier. Moreover, the model of transaction is proposed to be formated by observing a few principles. Then the earlier detection of non-serializable transaction can be performed more efficiently on this well formated model.

Secondly, because the overhead of characterizing the acyclicity of DSISG will grow unacceptable when the number of concurrent transactions increase significantly, I propose a mechanism which makes it more efficient to detect nodes that may cause a cycle in DSISG. A new hierarchical structure of DSISG is introduced. Two variations of algorithm that manages this hierarchized graph are provided. The tradeoff between efficiency and precision are discussed as well.

In this thesis I mainly focus on solving the above problems theoretically. It should be acknowledged that no real implementation of these solutions is included in this thesis. In the future, experiments will be made to confirm the theoretical evaluation of solutions presented in this thesis.

The thesis is structured as follows:

Chapter 1 provides a brief description of the main problems and a strategy for solutions thereof.

Chapter 2 introduces the concept of database concurrency control. Computational models of database transaction are presented. The correct criterias of concurrent execution of transactions are also given.

Chapter 3 presents numbers of basic concurrency control algorithms from two categories.

Chapter 4 studies Snapshot Isolation protocol. Defect of Snapshot Isolation protocol is pointed out after describing the algorithm. Previous solution by other researcher on solving this defect is also presented.

Chapter 5 discusses the limitation of the solution mentioned in chapter 4. An approach called DSISG is elaborated.

Chapter 6 proposes the segmented model of database transaction, which makes DSISG more implementable.

Chapter 7 describes how to minimize the system overhead while preserving the serializability of Snapshot Isolation. Self-adjusting acyclic graph is presented and algorithm that manage self-adjusting graph is studied.

Chapter 8 summaries the contributions of this thesis and formulates some open problems.

Chapter 2

Database Concurrency Control

The ability of concurrently handling the tasks submitted by the different users is one of the main requirements imposed on the modern database systems. Database concurrency control deals with the issues arising when the users simultaneously process shared data. The main objective of concurrency control algorithms is to find a correct and efficient synchronization of concurrent processes accessing the shared database resources. Protocols, criteria and efficiency issues of the database concurrency algorithms control have been studied by the researches in recent twenty years([6],[12], [25], [2], [3]). This chapter reviews the basic concepts of concurrency control in database systems. The definition and the intuitions related to a concept of database transaction are presented in section 1.1. Section 1.2 presents the evaluation criteria of concurrency control protocols. A conceptual model of a database system used in this thesis is included in section 1.3.

2.1 Database Transaction

A database transaction is an execution of a program submitted by a user accessing a shared database. The transactions retrieve and modify data. Logically, a transaction consists of numbers of read operations and write operations(include insert, update and delete). A boundary of transaction should is marked by a start and a commit(abort) of program. Because numbers of transactions may access same data simultaneously, the following features should be attached to database transaction to ensure the consistency of database content. In order to preserve database consistency the transactions must satisfy the properties listed below.

1. Atomicity:

   From the perspective of a user, a transaction is executed completely or not at

all. Transaction is ended with "commit" or "abort". At "commit" point a transaction completes successfully without any errors. At "abort" point a transaction is canceled and a database is automatically brought back to a state it was before the transaction started. In this situation, the database appears that the transaction had never been executed at all. The same actions are performed when a transaction fails due to the events like hardware corruption, operation system failure etc.

2. Consistency:
   The consistency of a database system is preserved by enforcing the logical consistency constrains.A transaction must take a database from one consistent state to another. For example, the debit record and credit record of one customer in a bank's database should be mutually consistent. When a transaction is about to commit, the constraints like the one listed above must be satisfied. However, while a transaction is processed, the temporally inconsistent states are tolerated and unavoidable.
   Normally, the enforcement of database consistency is implemented by checking constraint . At a certain stage of transaction execution, the logical consistency constraints will be verified on an updated database. If the verification fails then transactions reaches its "abort" pint and it is automatically rolled back. On the other hand, because of the properties of transactions, many consistency constraints do not have to be checked. For example, for a constraint requiring a quantity to be positive, there is no need to check this constraint after the execution of a transaction which increases that quantity.

3. Isolation:
   A transaction is isolated from the other transactions. It means that transactions do not communicate one with each other and transactions can only operate on a consistent state of a database. Only the results of committed transactions are "visible" to the other transaction. This property hides the concurrent executions of the transactions from the database users. A sufficient condition for isolation is that the concurrent executions of the transactions are equivalent to the sequential ones.

4. Durability:
   The results of committed transactions must remain permanent in a database. The modifications to a database should be able to survive the software or hardware

failures.

Summarizing, an execution of a transaction is a sequence of read and/or write operations and it should end with either "commit" or "abort". ACID properties guarantee the consistency and security of database system accessing by concurrently running transactions.

2.2 Transaction model and conventions in concurrency control

2.2.1 Transaction Application

Typically, a database application is a computer program that consists of a mixture of SQL statement and the statements of a general purpose programming language(such as JAVA or C, etc). The database applications submitted by the users are executed by a database system as the database transactions. The following example shows a sample database application and its sample execution traced as a sequence of elementary operations forming a database transaction.

Example 2.2.1

Then debit/credit transaction can be considered the most popular transaction example and has become the basis of TPC-A, TPC-B benchmarks which measure the performance in database environments typical in transaction processing applications.([23]) Suppose there is a user who want to transfer a mount of money from account A to account B, the following application will handle this request.

```
Procedure transfer(accA in number,
                   accB in number,amount in number)
Declare
    accB_balance number;
    accA_balance number;

  Begin
    select balance from acc into accA_balance
      where acc_number=accA;
    accA_balance := accA_balance-amount;
    if accA_balance < 0 then                    /*Statementα*/
```

```
            abort;
        else
            begin
                update acc set balance=accA_balance
                    where acc_number=accA;
                select balance from acc into accB_balance
                    where acc_number=accB;
                accB_balance := accB_balance+amount;
                update acc set balance=accB_balance
                    where acc_number=accB;
                commit;
            end;
        end if;
end;
```

To study the synchronization of database transaction we do not need to consider all details of the application listed above. A sequence of operations on data items is completely sufficient. For example, the execution of the application listed above may results with the following sequence of operations on data items:$R(a)R(b)W(a)W(b)C$. In the following sections, variant computational models of database transactions will be formally presented.

2.2.2 Page Model

The *page model* of transactions was the subject of theoretical studies since paper [17] and [4]. Now, it is the widely accepted by researchers as the conventional model of database transaction. From an execution trace of the sample application above we find that all higher-level operations can be mapped to read operation(select statement) and write operation(insert or update statement) on pages(also known as blocks).In the page model a database consists of a finite set of data items. The data item may be thought as pages as these are the elementary items involved in read and/or write operations. When an application is executed, a sequence of operations is submitted by the application to a database server. From a database server side a sequence of executed operations is considered as a transaction. If in an application given in example 2.2.1 a return value of statement α is true, then a transaction recognized by a database server is:
$T_1 : R(accA\_balance)$ *abort*

Otherwise, if the return value of α is false, then a transaction is formed by the following operations:
$T_2 : R(accA\_balance)\ \ R(accB\_balance)\ \ W(accA\_balance)\ \ W(accB\_balance)\ \ commit$
Furthermore, we find that operations in page model of transaction are not necessary to be totally ordered. As long as ACID principles apply, the order in which two or more operations are executed does not matter. For the transaction T_2 above, the final result in a database will be no different if the order of operations is changed to:
T_2‘ : $R(accA\_balance)\ \ W(accA\_balance)$
$R(accB\_balance)\ \ W(accB\_balance)\ \ commit$
A page model of transactions is defined as follows. ([25]).

Definition 2.2.1 *Page Model Transaction*
A transaction is a pair

$$t = (op, <)$$

where op is a finite set of steps of the form r(x) or w(x), $x \in D$, and $< \subseteq op \times op$ is a partial order on set op for which the following holds: if $p, q \subseteq op$ such that p and q both access the same data item and at least one of them is a write step, then $p < q \vee q < p$.

Therefore, in the page model of transactions, a read and write operations on the same data item, or two write operations on the same data item, must be ordered.

In a database application, when a data item is read its value is saved in a local variable. The value of the variable will be available for any possible further operation on the same data item until transaction is completed. Moreover, only the last write step determines the final value of data item produced by the transaction. Any other write operations on the same data item will be overwritten. On the other hand, before a data item x is written, the new valued will be computed and stored in a local variable as long as the transaction is still running. Then, after the write operation on x, the new value can be accessed from the local variable. Any further read of x from the database will lead to an unnecessary overload. Therefore, the following additional rules enhance the page model of transactions:

- A transaction reads or writes a data items at most one time.
- No data item is read after it has been written.

In the page model of transactions, the concurrent execution of a set of transactions can be mapped into a sequence of operations. Such sequence is called as a *schedule*

[25]. A software that ensure the correctness of a schedule by applying an implemented concurrency control technique is called as a *scheduler*.

Definition 2.2.2 *Let $T = t_1, t_2, \cdots$ be a limited set of transactions. Each $t_i \in T$ has the form t_i = (op_i, $<_i$) with op_i denoting the set of operations of t_i and $<_i$ denoting their ordering, $1 \leq i \leq n$*
A schedule for T is a pair s=(op(s), $<_s$) such that:

1. *s consists of the union of the operations from the given transactions plus a termination operation, which is either a c_i(commit) or an a_i(abort), for each $t_i \in T$;*
2. *for each transaction, there is either a commit or an abort in s, but not both;*
3. *all transaction orders are contained in the partial order given by s;*
4. *the Commit or Abort operation always appears as the last step of a transaction;*
5. *every pair of operations $p, q \in op(s)$ from distinct transactions that access the same data item and have at least one write operation among them is ordered in s in such a way that either $p <_s q$ or $q <_s p$*

2.2.3 Object Model

The object model of transactions has been proposed in [1]. It provides a framework for operations on arbitrary types of objects. Additionally, it is possible in the object model to clearly describe the case where an transaction is nested and called by other transaction.

Definition 2.2.3 *Object Model Transaction*
A transaction t is a finite tree of labeled nodes with

- *the transaction identifier as the label of the root node,*
- *the names and parameters of invoked operations as labels of inner(i.e., non-leaf, non-root) nodes,*
- *page model read/write operations as labels of leaf nodes, along with a partial order "<" on the leaf nodes such that for all leaf node operations p and q with p of the form w(x) and q of the form r(x) or w(x) or vice versa, we have $p < q \vee q < p$*

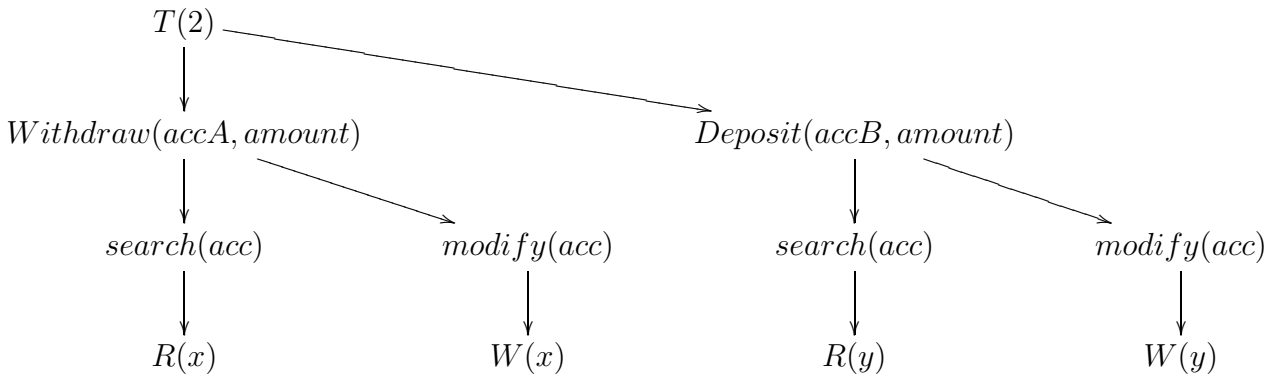


Figure 2.1: Example for object model transaction

For example, the object model of transaction T(2) in section 2.2.2 is presented in figure 2.1. Compare with the page model, object model is an enriched alternation page model. The semantics and business logic inside the application is shown explicitly in object model. In figure 2.1, we can see transaction T(2) logically consists two part:withdraw and deposit. Before operations access the page in database in leaves level, the accessing of table "acc" is shown in level 3. This object model of transaction presents rich information such as semantics, operations and object(such as table, index) be accessed in transaction,such that implementation performance of concurrency running transactions can be acquired.

2.2.4 Semantic Model

The semantic model of database transactions has been proposed in [3].The model allows for the explicit demonstration of the preconditions and postconditions in a transaction. Unlike listing the operations on various data items, the semantic model present the semantics of transaction application. The semantics of a transaction, T_i, can be formally characterized by the triple:

$$\{I_i \wedge B_i \wedge (\overline{x}_i = \overline{X}_i)\}T_i\{I_i \wedge Q_i\}$$

where I is a logical consistency constraint imposed on the contents of a database, and I_i is the conjunction of I required for the correct execution of T_i.For example, in the application given in example 2.2.1, the consistency constraint is: *the balance of all accounts can not be negative*. Consequently, I_i is both the precondition and the postcondition of T_i. B_i describes all conditions that T_i assumes to be true of the arguments passed to it. In the example 2.2.1, amount is the parameter representing the

money to be transfered from account A to account B, then B_i should assert $amount \geq 0$. Q_i is the result and asserts that T_i has achieved its purpose. In the same example, if T_i transfer the money successfully, the final balance of account A is less than initial balance while the final balance of account B is more than initial one. In semantic model, the initial value of database variable is recorded by $\overline{X}_i$. $\overline{x}_i$ holds the value that changed by transaction T_i. Then, the semantic model of transaction in example 2.2.1 is:

$\{accA.balance \geq 0 \wedge accB.balance \geq 0 \wedge amount \geq 0$
$\wedge accA.balance = BAL1 \wedge accB.balance = BAL2\}$
T_i
$\{accA.balance \geq 0 \wedge accB.balance \geq 0 \wedge accA.balance = BAL1 - amount$
$\wedge accB.balance = BAL2 + amount\}$

The execution of a transaction is correct if the first and third part of semantic model is true(i.e. semantic correct). Likewise, the concurrent execution of transactions is correct if all transactions are semantic correct. However, the critical defect of semantic model is, it is hard to be implemented. The semantic modelization of transaction can not be fulfilled as easy as the other two models. The analysis of interference between semantics of transactions is also hard to be realized by computer while it can be easily implemented by locking mechanism in page model. Therefore, there are not many researchers study database concurrency control by following semantic model since it has been proposed.

2.2.5 Transaction model used in this thesis

In the former three sections, I introduced three different computational model for database transaction. Page model and object model belong to one family while semantic model belongs to another. Page model and object model describe the transaction by abstractly presenting the detail of operations and data items involved in transaction. Then the correctness of concurrently running transactions can be assured by solving the conflicts between operations. The difference between page model and object mode is object model contains more information like accessed tables or indexes. Moreover, nested transaction can be described clearer by object model. On the other hand, semantic model ignores the specific details of operations and on which data items these operations are executed. Preconditions and postconditions of transaction are elaborated in semantic model by applying Hoare's logic([14]). When preconditions and postconditions are true we say that the execution of transaction is semantic correct.

Then the concurrent execution of transactions can be guaranteed if every single transaction is semantic correct.

In the scope of this thesis, the complicated analysis of semantics or nested transactions will not be included. The fundamental and algorithms of concurrency control can be clearly expressed and easier understood in the simple page model. More importantly, one of my solutions, "segmented transaction model" which will be presented in chapter 6 is also based on page model. Consequently, only page model will be used as conventional transaction model in this thesis.

2.3 Correct execution of concurrent transactions

As stated above, the main objective of database concurrency control is to avoid the incorrect execution of concurrent transactions. So, the formal definition of correctness is necessary for the study on concurrency control algorithms. After the introduction of the concept of database transaction and its computational model, this section will focus on the page model of transactions and discuss the correctness for their concurrent execution.

2.3.1 Typical concurrency problems

Because database transaction can be executed concurrently, some data items might be accessed by some operations belong to different transactions. We say two operations conflict with each other if they are on the same data item and at least one of them is write operation. In the absence of proper concurrency control, conflicting operations may breach the ACID of particular transactions. The following example is a typical concurrency problem is known as "dirty read".

Example 2.3.1

T_1: write(x) abort

T_2: read(x) write(x) commit

X is a bank account. The balance of x is not allowed to be negative and it is 100$ at the moment. Suppose Mr. and Mrs Smith all have the privilege to access account x. One day Mr. Smith starts a transaction T_1 while Mrs. Smith starts another one T_2 at the same time. Firstly, T_1 deposits 100$ into x, so balance(x) becomes 200$ temporally. Then, T_2 try to withdraw 200$ from x. Before this operation can be executed, T_2 will

read x to check whether it has enough money. Because of the absence of concurrency control, T_2 will get the value that created by T_1, which is 200\$. Then the request of withdraw is approved, T_2 commit. After that, Mr. Smith will get his 100\$ back by aborting that deposit transaction T_1. The result is, the balance of x becomes negative.

The reason of the database corruption above is, there is no proper concurrency control on the operation which tries to read an uncommitted(dirty) data item. The uncommitted modification on data item might be discarded later by aborting corresponding transaction. Then the consistency of database may be broken by some operation based on the value of dirty read data.

Dirty read is the concurrency problem which involves with one read operation and one write operation on the same data item. The next example will present an incorrectness which involves two write operations.

Example 2.3.2

T_1: read(x) write(x)commit

T_2: write(x) commit

x is still the bank account that can be accessed by Mr. and Mrs Smith simultaneously. The balance of x is 100\$. T_1 and T_2 are started by the couple individually at the same time. Firstly, T_1 tries to withdraw 100\$ from x. After read(x) is executed to check whether the money in x is enough to be withdrawn, T_2 deposit 100\$ into x. Balance(x) is updated as 200\$ temporally. Then request of withdraw in T_1 is approved. Because the unawareness of the modification made by T_2 on x, T_1 will update and commit the balance of x as zero. At last, the balance of x will still be zero after T_2 is committed. Then, Mr. and Mrs Smith will lost 100\$ and the bank encounters an inconsistency between ledger and cashes.

In the example above, the modification on x made by T_2 was overwritten by T_1, i.e. the T_2's update is lost. The isolation property of T_2 was violated by this lost of update. That violation leads to the incorrect execution of T_1 and T_2. What can be seen from the above examples is that the data accesses performed by concurrently executing transactions have a potential of conflicting with each other. Therefore, some form of concurrency control has to be taken to ensure the correct execution of concurrent transactions. In examples above, the correctness criteria is the common sense of bank

business. However, for the research work on concurrency control of transactions, a general criteria of correctness must be defined formally. The next chapter will manifest numbers of classical correctness criteria of concurrent execution of transactions.

2.3.2 Serializability of concurrent execution

In the preceding examples, the errors were caused by the interleaved execution of operations from different transactions. To avoid these and other problems, the kinds of interleavings between transactions must be controlled. One way to avoid interference problems is not to allow transactions to be interleaved at all. For a pair of transaction, if one transaction is not allowed to start until the transaction before it has been explicitly committed or aborted, the execution of these transactions is called serial. Serial executions are correct because each transaction individually is assumed to be correct, and transactions that execute serially cannot interfere with each other. However, if DBMS is forced to process transactions serially, it may make very insufficient use of its resource. DBMS will become inefficient without the concurrency. In order to acquire a DBMS which can handle transactions efficiently and correctly, we can include other executions of the same set of transactions as long as they have the same effect as serial ones. Such executions are call serializable. An execution of set of transactions is serializable if it produces the same output and has the same effect on database as one possible serial execution of the same transactions. Since serial execution is always correct, and since serializable execution has the same effect as a serial execution, the correctness of serializable execution is self-proved.Look back at "lost update" presented in example 2.3.2, the execution is incorrect because it is not serializable. Two possible serial execution of two transactions in example 2.3.2 is $T_1 \rightarrow T_2$ or reverse. The result of these two serial executions are all $balance(x) = 100\$$. However, the result of execution in example 2.3.2 is balance(x)=0 which does not equal to the result any possible serial execution. So, this execution is not correct(serializable).

Compare with the general concept of serializable mentioned above, view serializable(proposed in [27]) is a specific criteria of serializability and especially useful for the formal treatment of concurrency control algorithms for multiversion data.

Definition 2.3.1 *Concurrent execution of database transactions is view serializable if there exists a possible serial execution of the same set of transactions such that in both executions each transaction reads the same values and the final states of the database are the same.*

However, the complexity of testing the view serializability of a schedule is proved to be NP complete. It means that it takes to much time to check whether execution of a database operation violates view serializability correctness criterion. So, due to the high complexity of its recognition problem, view serializability is inappropriate as a correctness notion for practical scheduling algorithms. So, conflict serializability is proposed.

Another notion of serializability , conflict serializability, is the most important for the practice of database concurrency control. It is computationally easy to test and differs significantly from view serializability. Conflict serializability is based on a notion of conflict that was briefly mentioned in section 2.3.1. Let s be a schedule, T_i and T_j are different transactions belong to this schedule. We say two operations $p \in T_I$ and $q \in T_j$ are conflict with each other in s if they access the same date item and at least one of them is a write. $conf(s) = \{(p, q)|$p, q are in conflict in s and $p <_s q\}$ annotates the conflict relation of s.

Definition 2.3.2 *Concurrent execution of database transactions is conflict serializable if there exists a possible serial execution of the same set of transactions such that in both executions the order of conflicting operations is the same.*

For the schedule which caused "lost update" in example 2.3.2,

$$conf(s) = \{(r_1(x), w_2(x))(w_2(x), w_1(x))\}$$

and this conflict relation can be obtained from the serial execution $T_1 \rightarrow T_2$ or $T_2 \rightarrow T_1$. Therefore, schedule in example 2.3.2 is not conflict serializable.

The most important difference between conflict serializable and the previously introduced view serializable is that the former can be tested efficiently. The conflict serializability of a schedule can be characterized via the corresponding serialization graph. In serialization graph, nodes represent committed transactions. Directed edge from transaction T_i to transaction T_j indicates that there are operations $p \in T_i$ and $q \in T_j$ such that p and q are in conflict. The order of conflicting operations is represented by the direction of edge. Consider the following schedule:

$$s = R_1(y)R_3(p)R_2(y)W_2(y)W_3(x)R_2(q)C_2W_3(q)C_3W_1(p)C_1$$

Conflicts $(R_1(y), W_2(y))$ and $(R_2(q), W_3(q))$ mean that in a serial execution $T_1 < T_2 < T_3$. However, conflict $(R_3(p), W_1(p))$ says that T_3 should be prior to T_1 in a serial

execution. Obviously, those are impossible to meet simultaneously. The dilemma is represented as a cycle in serialization graph of s(shown in Figure 2.2, which means s is not conflict serializable. In [9], the theorem widely known as "conflict serializability

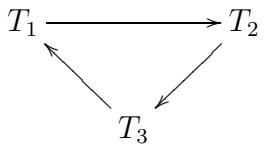


Figure 2.2: The serialization graph of a non conflict serializable schedule

theorem" is defined:

Definition 2.3.3 *Let s be a schedule. Then s is conflict serializable if and only if serialization graph SG(s) is acyclic.*

Compare with view serializability, conflict serializability is more restrictive. A schedule which is conflict serializable is also view serializable but a view serializable schedule is not necessary also conflict serializable. More concurrency is available if a scheduler is based on view serializability. From a practical point of view, the complexity of testing the view serializability of a schedule s is much higher than that of the conflict serializability of s. Consequently, most known single-version concurrency control algorithms are conflict serializability based. Their goal is to order conflicting operations in a consistent way. The result of concurrent execution under these algorithms produce only conflict serializable. On the other hand, as stated previously, the concept of view serializability is very useful for multiversion concurrency control algorithms. An alternation of criteria especially for multiversion called One-copy serializability will be elaborated in the next chapter.

In addition to view serializable and conflict serializable, other correctness criterias were also studied by researches. Some are more restricted than conflict serializable such as order-preserving serializable, recoverable execution, cascadeless execution which were proposed in [24], [7]. Some like partial order serializability, predicatewise serializability ([15]) are proposed for the implementation of particular database on which CAD or CAD-like applications are running. Moreover, a criteria which is based on the semantic of transactions are proposed in [3]. In this thesis, all the concurrency control mechanisms will focus on view serializability and conflict serializability.

Chapter 3

Concurrency Control Algorithms

Database concurrency control algorithms are categorized into two types of algorithms depending on a way the algorithms handle the conflicting operations. A group of concurrency control algorithms called as *pessimistic protocols* blocks the execution of an operation that conflicts with an operation executed earlier by another transaction. A group of concurrency control algorithms called as *optimistic protocols* never blocks the execution of an operation. The verification of conflict serializability criterion is performed in certain frequency, normally it is only once when transaction is about to commit. When the verification fails a transaction that caused a non-serializable execution is forced to abort. This chapter reviews the major database concurrency control algorithms that belongs to pessimistic and optimistic protocols.

3.1 Pessimistic Protocol

The simultaneous access to the shared data items is managed by the locks set on and removed from the data items on behalf of a transaction. A data item locked by a transaction is not available to other transactions until a lock is released. Before a transaction T is allowed to access a data item x, it has to request a lock on the data item. Next, the scheduler checks whether x has been locked by another transaction or not. If a lock has been set on x on behalf of another transaction then transaction a request to grant the lock is suspended and T has to wait till data item x is unlocked. If a data item x is not locked than the requested lock is granted to a transaction T, x is locked by the scheduler on behalf of the transaction, and intended operation is performed by a transaction T on a data item x. A transaction T releases a lock on x when access to x is no longer needed. Then the scheduler checks whether there is another blocked transaction in a queue of transactions for the lock on x. If it is, transaction in the head of queue will obtain the lock and resume. All the pessimistic protocols are based on the locking mechanism described above. Different kinds of locks

are managed in the different ways by different algorithms.

3.1.1 Two Phase Locking

Two phase locking is the most frequently implemented concurrency control algorithm in the commercial database systems. Based on page model of transaction, two kinds of locks are involved in two phase locking. When a data item x is read or written by an operation which belongs to transaction T, a read lock(rl)or write lock(wl) will be put on x on behalf of T. Similar to the way operations on the same date item conflict in page model, different locks on the same data item will also conflict with each other in certain way. The table below shows the compatibility between read and write lock.

	Read Lock(x) Hold by T_i	Write Lock(x) Hold by T_i
Read Lock(x) Request by T_j	Granted	if i $\neq$ j Rejected if i = j Granted
Write Lock(x) Request by T_j	if i $\neq$ j Rejected if i = j Granted	if i $\neq$ j Rejected if i = j Granted

Figure 3.1: Compatibility of Locks in Two Phase Locking

Two locks $pl_i(x)$ and $ql_j(y)$ are in conflict if x=y, $x \neq j$, and operations p and q are in conflict(i.e., at least one of them is write operation). The requested lock issued by transaction T_i will be blocked by the conflicting lock that has already been held by other transactions T_j. This lock can be granted to T_i when T_j releases the conflicting lock. Therefore, read lock is also called shared lock and write lock is called exclusive lock. On the other hand, if x=y and i=j, the requested lock will be granted anyway. Transaction running under two phase locking protocol consists of two phases: lock acquiring phase and lock releasing phase. Each transaction must acquire all locks before it is terminated. No lock held by a transaction T could be released until T is terminated. In the following, an example will be given to present how two phase locking works.

Example 3.1.1

Suppose transactions T_1, T_2, T_3 are running concurrently under two phase locking. The submission order of operations is shown below:

T_1 : $R_1(x)$ $W_1(x)$ $W_1(z)C_1$

T_2 : $R_2(y)$ $R_2(x)$ $W_2(x)C_2$

T_3 : $R_3(z)$ $W_3(y)$ C_3

Then the execution of this schedule can be mapped into the following history:

$RL_3(z)$, $R_3(z)$, $RL_1(x)$, $R_1(x)$, $RL_2(y)$, $R_2(y)$, $WL_1(x)$, $W_1(x)$,
$WL_3(y)$ *blocked,* $RL_2(x)$ *blocked,* $WL_1(z)$ *blocked,* $WL_2(x)$ *blocked,* C_3,
$RL_3(z)$ *released,* $WL_1(z)$, $W_1(z)$, C_1, $WL_1(x)$ *released,* $WL_1(z)$ *released,*
$RL_2(x)$, $R_2(x)$, $WL_2(x)$, $W_2(x)$, C_2

Obviously, because some of the operations have been suspended by locking mechanism, the concurrent execution of the transactions under two phase locking is equivalent to the following serial execution: $T_3 < T_1 < T_2$. Consequently, the order of conflicts between operations is preserved. The schedule in example 3.1.1 is conflict serializable. More intuitively, the equivalent serial execution of schedule s under two phase locking can be described by serialization graph(s).

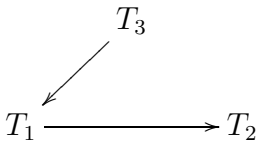


Figure 3.2: Serialization graph of schedule in example 3.1.1

Under two phase locking, transaction will not release the locks it holds until all locks it requires are granted. So, it is possible that a group of transactions will be suspended forever if each of them requires conflicting locks that have already been held by the others. For example, in the simplest case with only two transactions T_1 and T_2, T_1 requests a conflicting lock that T_2 is holding so that T_1 is waiting for T_2 to releas it, meanwhile, T_2 requests a conflicting lock that T_1 is holding so that T_2 is waiting for T_1. Then T_1 and T_2 will be blocked by each other and no transaction is able to go on. This defect of two phase locking algorithm is called "deadlock". A straightforward way to solve this problem uses a timeout period. Once a transaction T is waiting for a lock longer than for a timeout period, the scheduler assumes that T is involved in a deadlock and aborts the transaction. This technique is not completely reliable as the correct recognition of a deadlock depends on the length of timeout. Longer a timeout period is, more probable it is to recognize a deadlock correctly. On the other hand, a shorter timeout period implies less time time spent by the transactions in an idle state. So, the tunning of timeout period becomes tricky and it can not be guaranteed that there is not "innocent" transaction is aborted by this deadlock guessing approach.

Another solution to deadlock problem is to construct and to maintain a *wait-for* relationships while the transactions are running. The wait-for relationship between conflicting operations can be revealed by wait-for graph(WFG). WFG is a graph G =(V,E) whose nodes are the active transactions, and in which an edge of the form(T_i,T_j) indicates that T_i waits for T_j to release a lock that it needs. Obviously, a cycle in wait-for graph reveals a deadlock. As an example consider the following concurrent execution of three transactions controlled by two phase locking scheduler.

Example 3.1.2

T_1: $R_1(x)$ $R_1(y)$ $W_1(y)$ $R_1(z)$ C_1

T_2: $W_1(x)$ $R_2(p)$ $W_2(p)$ C_2

T_3: $R_3(q)$ $W_3(z)$ $R_3(p)$ C_3

In this schedule, T_2 is waiting for T_1 to release lock on x; T_3 is waiting for T_2 to release lock on p; T_1 is waiting for T_3 to release lock on z. The wait-for graph of this execution is:

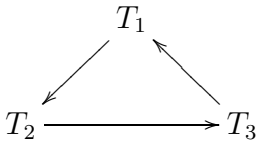


Figure 3.3: Wait-for graph of schedule in example 3.1.2

A cycle in a wait-for graph indicates a deadlock

When a deadlock is discovered by the scheduler, one involved transaction will be aborted to break the deadlock. Then, the scheduler releases all locks held by the aborted transaction and the transactions waiting for these locks can be continued.

3.1.2 Some variants of two phase locking

In addition to the original two phase locking(2PL) algorithm, many other pessimistic algorithms based on a similar idea have been proposed in the past. Conservative 2PL is a more restricted version of two phase locking in a way that it eliminates deadlocks at the expense concurrency level. A transaction controlled by a conservative 2PL scheduler is not allowed to continue until all locks it needs are granted at a start point. It needs a transaction T to declare its entire read, write sets in advance. If some the requested locks are already held by the running transactions then T has to wait. This

is why a blocked transaction never holds a lock and because of that two transactions will never block each other. In [25], the definition of this conservative two phase locking is presented as:

Definition 3.1.1 *Under conservative 2PL each transaction sets all locks that it needs in the beginning, i.e., before it executes its first r or w step. This is also known as preclaiming all necessary locks up front.*

Altruistic locking(AL) protocol proposed in [18] is an extension of two phase locking protocol motivated by the inefficient performance of when dealing with the long transactions. A long transaction is a transaction that has much longer execution time when compared with the ordinary transactions. Transactions under two phase locking will not release locks it held unless all required locks are granted. The transactions waiting for the items locked by a long transaction are delayed for a long period of time. So, the performance problem of 2PL is serious when dealing with long transactions.

Under altruistic locking protocol, the transactions are allowed to hold conflicting locks on a data item simultaneously under certain conditions. Besides set lock and release lock, AL protocol introduces a third access control operation called as *donate*. The *donate* operation is used to notify the scheduler that the transaction will no longer access a data item that has been locked by it. Then the locked data item can be "donated" to other transaction which requires an access on it. The donating transaction is allowed to put lock on other data items in the future. Lock and donate operations do not need to follow a two phase rule. The formal definition of altruistic locking is presented in the paper [18]:

Definition 3.1.2 *A scheduler is an altruistic locking scheduler if it is a 2PL scheduler and obeys the following four rules:*

1. *Items can not be read or written by T_i once it has donated them;*
2. *Donated items are eventually unlocked;*
3. *Transactions cannot hold conflicting locks simultaneously, unless one has donated the data item in question;*
4. *If a transaction T_j locks a data item that has been donated (and not yet unlocked) by another transaction, we say that it is in the wake of the donating transaction T_i. T_j is indebted to transaction T_i in a schedule s if*

- *$o_i(x)$, $d_i(x)$(the donating on item x by transaction T_i), $p_j(x) \in$ op(s) such that $p_j(x)$ is in the wake of T_i and*
- *either $o_i(x)$ and $p_j(x)$ are in conflict or some intervening operation $q_k(x)$ such that $d_i(x) <_s q_k(x) <_s p_j(x)$ is in conflict with both $o_i(x)$ and $p_j(x)$*

When a transaction T_j is indebted to another transaction T_i, T_j must remain completely in the wake of T_i until T_i begins to unlock items.

Similar to conservative 2PL, altruistic locking requires the read set and write set of a transaction to be provided before the transaction starts. Altruistic locking does not solve the long transaction problem completely although it reduces the likelihood of transaction's long period delaying. It is still possible that some transactions suffer long time waits under altruistic locking. In the next section, algorithms belong to another family will be introduced. The simultaneous accesses to same data items will never be delayed by optimistic algorithms. Different approaches but not locking are used by optimistic algorithms to ensure the serializability.

3.2 Optimistic Protocol

In this chapter, the concept and features of long transaction will be introduced at first. Then optimistic algorithms will be studied from two classes: single-version algorithms and multi-version algorithms.

3.2.1 Long Transactions

A long transaction has a longer duration than regular transaction. Long transaction may access many data items, perform lengthy computations, pause frequently for feedback of users, etc. For example, in a bank system, the transaction which computes the interests of all accounts for entire last year can easily take several hours. Similarly, a transaction that reads/writes large file(a few gigabytes) will also take a significant amount of time. Because pessimistic algorithms such as two phase locking and conservative 2PL require lock to be kept for the whole duration of transactions, the performance of system might be badly harmed by the delaying of transactions.

Optimistic algorithms allow transactions to be executed concurrently without delaying. When a transaction is about to commit, some validation processing will be taken to ensure the serializability of the whole schedule. If validation succeeds, the

transaction commits. Otherwise, the transaction will be aborted and restarted. So, to a system in which most transactions are long transactions, optimistic algorithm is more efficient than algorithms based on locking.

3.2.2 Serialization Graph Testing

As discussed in section 2.3.2, serialization graph is an useful graphical approach that can be used to characterized the conflict serializability of schedule. The concurrent execution of a set of transactions is conflict serializable if and only if corresponding serialization graph is acyclic. In [8], Serialization Graph Testing(SGT) was developed as an optimistic algorithm. According to the original definition of serialization graph, a serialization graph contains nodes for all committed transactions and for no others. Such SG differs from the one that is maintained by SGT scheduler. In SGT scheduler's serialization graph, only node for active transaction will be included. A node will be added in or removed from the graph when corresponding transaction is started or committed. A different term, stored serialization graph(SSG) is used to denote the SG maintained by a SGT scheduler. The conflict serializability of execution is guaranteed by preserving the acyclicity of stored serialization graph.

When an operation $p_i(x)$ of transaction T_i arrives and $p_i(x)$ is not commit, it can be executed anyway. If $p_i(x)$ is the first operation of T_i, the scheduler creates a node for T_i in current serialization graph G. Then conflict edge will be inserted in G if $p_i(x)$ conflicts with others. On the other hand, if $p_i(x)$ is commit, the cyclicity of G will be evaluated. If G is acyclic, T_i is allowed to commit. If G is cyclic, transaction T_i will be forced to abort. The node for T_i and all edges connects to it are removed from G. For a node stands for a committed transaction in serialization graph G, it can be removed from G as long as it is a source node(a node without outgoing edges). For example , consider the following schedule:

Example 3.2.1

These transactions are running concurrently under Serialization Graph Testing:

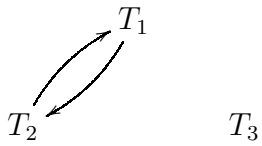
T_1 : $R_1(x)$ $W_1(x)$ $W_1(y)C_1$

T_2 : $R_2(y)$ $R_2(x)$ $W_2(z)C_2$

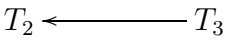
T_3 : $R_3(z)$ $W_3(z)$ C_3

Before C_1 can be executed the serialization graph is:

There is a cycle in the graph, so T_1 will be aborted, node T_1 will also be removed



from graph. When C_2 is arrived, the serialization graph is:



The acyclicity of graph means T_2 is able to commit. Because node T_2 is not a source node, it can not be removed from graph. The serialization will still be remained as above. When C_3 is arrived, transaction T_3 will be allowed to commit. Because node T_3 is a source node and stands for a committed transaction, it can be removed from graph. After that, node T_2 becomes a source node. So it can be removed too.As the result, by aborting T_1 which violates the conflict serializability, a serializable execution of T_2 and T_3 is preserved.

Serialization Graph Testing is such an intuitive and important algorithm that most other graphical concurrency control mechanisms are based on it. In the future part of this thesis, number of graphical mechanisms which are similar with SGT will be studied more specifically.

3.2.3 Time Stamp Ordering

Time Stamp Ordering is from [22]. The scheduler assigns to each transaction T_i a unique timestamp ts(T_i). Normally, the value of time stamp is from a counter that increases every time a new time stamp is needed. The timestamp of a transaction is inherited by every operation of that transaction. So, the timestamp of an operation $o_i(x)$ is simply also the timestamp of transaction T_i that issues $o_i(x)$. Under Time Stamp Ordering, the order of conflicting operations are decided by their timestamps. More precisely, a time stamp ordering scheduler observes the following rule:

if $p_i(x)$ and $q_j(x)$ are conflicting operations and $ts(T_i) < ts(T_j)$, $p_i(x)$ must be executed prior to $q_j(x)$.

When operation $p_i(x)$ arrives after a conflicting operation $q_j(x)$ which has already been executed, if $ts(T_i) < ts(T_j)$, the approval of $p_i(x)$ will violate the rule above.

Therefore, this "too late" operation $p_i(x)$ will be rejected and transaction T_i has to be aborted. The behavior of Time Stamp Ordering algorithm is illustrated in the following example:

Example 3.2.2

$T_1 : R_1(x) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad R_1(p)C_1$

$T_2 : \qquad W_2(x) \qquad W_2(y)\ Abort$

$T_3 : \qquad\qquad R_3(y) \qquad\qquad\qquad W_3(x)C_3$

Because T_2 is started earlier than T_3, ts(T_2) < ts(T_3). But $W_2(y)$ arrives later than $R_3(y)$, so $W_2(y)$ is rejected such that T_2 is aborted.

Although transactions which don't have conflicts can be executed in arbitrary order under TSO, the actual execution order of conflicting transactions under TSO is strictly preserved as the starting order of these transactions. So, for a set of concurrent transactions, there exists only one equivalent serial execution under TSO. This is much more restricted than the concept of "serializable". Consequently, some transaction might be aborted unnecessary even if the schedule is conflict serializable. For example, in the next schedule, transaction T_1 will survive under 2PL or SGT, but it has to be aborted under time stamp ordering.

Example 3.2.3

$T_1 : R_1(p) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad R_1(x)Abort$

$T_2 : \qquad W_2(x) \qquad R_2(y)$

Beside to indicating the order of equivalent serial execution in timestamp ordering, the value of time stamps can also be implemented in other concurrency control algorithms. For instance, in chapter 4, time stamp plays an important role in the implementation of an efficient and widely used algorithm: Snapshot Isolation.

3.2.4 Multiversion Concurrency Control

In all the concurrency control algorithms mentioned above, each data item only has one copy in the database. This means, without any concurrency control, the result of one write operation might be overwritten by that of the other write operation. Therefore, algorithms introduced in sections above can be categorized as single-version algorithm. On the other hand, another kind of concurrency control algorithm, multiversion algorithm, keeps multiple copies for each data item. It is the optimistic algorithm which

will never block conflicting operations. In a multiversion concurrency control algorithm, multiple copies of data items will be kept simultaneous. The write operation on data item x will not overwrite the original value of x. On the contrary, a new version of x will be created. For each read(x), the scheduler will decide which version of x can be accessed. At particular point, normally is the commit point of each transaction, one version of each data item will be chosen by scheduler and saved into database permanently. Copies of data item x are annotated as x_i, x_j...where the subscript is the index of transaction that create the version. So, in a multiversion schedule, a write operation is always the form $W_i(x_i)$ while a read operation is the form $R_i(x_j)$(j might be equivalent to i). For multiversion concurrency control in general, there is no specific order of data versions is given. However, for some specific implementation of multiversion concurrency control, order of versions of data item x will be decided by "creating time"(the time on which a write operation is executed on x) or "committing time"(the time on which the created version is committed). To the same data item x, if a version x_i is ordered preceding to another one x_j, we say that the order of these two versions is $x_i \ll x_j$. The existence of multiple versions of data items should not be awarded of by transactions. They are only visible to the scheduler. On the perspective of outside user, the functionality of database system is still over the single version of data.

Because the concept of conflicting operations is based on the fact that there is only one version for each data item, conflict serializability is not applicable to be used as the correctness criteria of multiversion algorithm. The extension of view serializability, one copy serializability, has been introduced as the correctness criteria of multiversion algorithm in [5]. From the point of view of users, all the data stored in database system are kept single versioned. Recall that two schedules are view equivalent if they read the same values of data items and produce the same outputs. Similarly, a multivesion schedule and a single version schedule over the same set of transactions are view equivalent if they read the same vales of data items and produce the same outputs. One copy serializable was defined as the correctness criteria of multiversion algorithm in [5]:

Definition 3.2.1 *Let S be an Multiversion schedule over a set of transactions T, S is view equivalent to a serial, one version schedule over T if and only if S is one copy serializable(1SR).*

For instance, one single version schedule and one multiversion schedule are over the same set of transactions:

Example 3.2.4

Multiversion schedule:

T_1: $R_1(x_0)$ $W_1(x_1)$ C_1

T_2: $R_2(z_0)$ $W_2(y_2)$ $R_2(x_1)$ C_2

T_3: $R_3(y_0)$ $W_3(y_3)$ C_3

Single version serial schedule:

T_3:$R_3(y)$ $W_3(y)$ C_3

T_1: $R_1(x)$ $W_1(x)$ C_1

T_2: $R_2(z)$ $W_2(y)$ $R_2(x)$ C_2

In multiversion schedule, T_1 reads the original value of x; T_2 reads the original value of z and the version of x created by T_1; T_3 reads the original value of y. The final output of this schedule is $x = x_1$ and $y = y_2$. These inputs and outputs are exactly the same as that of the second single version serial schedule. So, the first multiversion schedule is one copy serializable.

In [5], the author proposed a mechanism which can characterize the correctness of general multiversion concurrency control algorithm. Serialization graph is modified to presented the order of versions of data items that were accessed by concurrent transactions. This serialization graph especially for multiversion algorithm is called Multiversion Serialization Graph(MVSG).

Definition 3.2.2 *For a given MV schedule S and a version order $\ll$, the multiversion serialization graph for S and $\ll$, MVSG(S,$\ll$), is serialization graph(S)(edges are issued by applying the definition of SG of single version schedule) with the following version order edges added: for each $R_k(x_j)$ and $W_i(x_i)$ where i, j, and k are distinct,*

1. *if $x_i \ll x_j$ then include $T_i \rightarrow T_j$;*
2. *otherwise include $T_k \rightarrow T_i$.*

Then, the one copy serializability of multiversion schedule can be characterized by the acyclicity of corresponding MVSG.

Theorem 3.2.1 *A multiversion schedule S is one-copy-serializable if and only if the multiversion serialization graph (MVSG) is acyclic.*

For example, the MVSG of multiversion schedule in example 3.2.4 is: The graph is acyclic, so this multiversion schedule is one copy serializable. Actually, the direction of

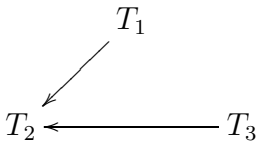


Figure 3.4: An example of Multiversion Serialization Graph

edges in MVSG clearly indicates the execution order of transactions in equivalent serial schedule. As it is revealed in Figure 3.4, any single copy serial execution of transactions in example 3.2.4 is equivalent to the multiversion concurrent execution as long as the following two conditions hold: (1) T_1 is prior to T_2. (2) T_3 is prior to T_2.

Sine MVSG actually is the classical serialization graph together with some additional edges caused by dependencies between different versions of the same data item, edges in MVSG can be categorized as the following:
Two types of edges in MVSG:

- A, Edges caused by conflict between operations on same versions of the same data item. Conflicting Operations belongs to different transactions.
- B, Edges caused by conflict between operations on different versions of the sam data item.Conflicting Operations belongs to different transactions.

In fact, only edges in category B are decided by two rules in definition 3.2.2. The first condition figures out the edge between two write operations from older version to newer version of the same data item. The second condition figures out the edge between a read operation on older version and a write operation on newer version of the same data item.

Chapter 4

Snapshot Isolation

Snapshot isolation [2] protocol is a multiversion concurrency control algorithm used by the database systems to enforce the logical consistency of a database while processing the database transactions. By implementing the techniques of multiversion and time stamps, Snapshot Isolation(SI) is free from concurrency problems such as dirty read, unrepeatable read, lost update, etc. SI protocol is free from the typical anomalies such as dirty read, unrepeatable read, lost update, etc. SI has been implemented in the systems including Oracle, PostgreSQL and Microsoft's SQL Server. In different systems, SI is implemented in different ways. For example, in Oracle, the mechanism "First-committer-wins" is replaced with a variation "First-writer-wins". Write/Read locks are used to obtain SI behavior. In this thesis, all the studies will be focus on the general concepts of Snapshot Isolation. This chapter presents a study of SI protocol.

4.1 Isolation levels

The problems with concurrent access to shared data are the subjects to the properties of concurrently running transactions. In some situations, the properties of the concurrently running database transactions are such that that the execution is free of a particular kind of concurrency problem. Then, an over restricted concurrency control algorithm may harm the performance of system. For example, consider a database system where no data item is inserted or deleted by a concurrent running transaction. Then, and algorithm, which puts long duration locks on the whole table accessed by the transaction is obviously over restricted. ANSI standard [26] defines three typical incorrect concurrent execution of transactions if there is no proper concurrency control. These incorrect executions are named "Phenomena". ANSI standard [26] introduced a concept of *isolation level*. Each isolation level can prevent the execution of transactions from experiencing particular phenomena. An overview of the relationship

between "phenomena" and isolation levels are presented in the following table.

Isolation Level	P1 Dirty Read	P2 Fuzzy Read	P3 Phantom
ANSI READ UNCOMMITTED	Not prevented	Not prevented	Not prevented
ANSI READ COMMITTED	Prevented	Not prevented	Not prevented
ANSI REPEATABLE READ	Prevented	Prevented	Not prevented
ANOMALY SERIALIZABLE	Prevented	Prevented	Prevented

Figure 4.1: ANSI SQL Isolation Levels Defined in the terms of phenomena

The descriptive specifications of phenomena are as follows.
Suppose the transaction T_1 and T_2 are running concurrently.

- Dirty Read: T_2 reads a data item x that has been written by T_1 before T_1 is committed.
- Fuzzy Read: T_1 reads a data item x. Before T_1 is committed, T_2 submits a write operation on x and commits.
- Phantom: T_1 reads a set of data items which satisfy a certain criteria C. Before, T_1 is committed, T_2 removes or inserts number of data items which also satisfy the same criteria and commits.

In [2] the informal definitions of isolation levels are criticized as being too ambiguous. The same work proposes a different set of phenomena that avoids the problems of the ANSI-SQL definitions. In addition new phenomena such dirty write, lost update, write skew$\cdots$, that paper more precisely redefines all the phenomena in the terms of the page model. The mathematical definitions of phenomena from [2] are shown below:

- Dirty Write: $W_1(\mathrm{x}) \cdots W_2(\mathrm{x}) \cdots$ (c_1 or a_1)
- Dirty Read: $W_1(\mathrm{x}) \cdots R_2(\mathrm{x}) \cdots$ (a_1 and c_2 in either order)
- Fuzzy Read: $R_1(\mathrm{x}) \cdots W_2(\mathrm{x}) \cdots$ ((c_1((c_1 or a_1) and (c_2 or a_2) in any order)
- Phantom: $R_1(\mathrm{P}) \cdots W_2(\mathrm{y\ in\ P}) \cdots$ ((c_1 or a_1) and (c_2 or a_2) in any order)
- Read Skew: $R_1(\mathrm{x}) \cdots W_2(\mathrm{x}) \cdots W_2(\mathrm{y}) \cdots c_2 \cdots R_1(\mathrm{y}) \cdots$ (c_1 or a_1)
- Lost Update: $R_1(\mathrm{x}) \cdots W_2(\mathrm{x}) \cdots W_1(\mathrm{x}) \cdots c_1$
  P is a set of data item which have to be read before executing a write operation on a particular data item(i.e. predicate)

- Write Skew: $R_1(\mathrm{x})\cdots R_2(\mathrm{y})\cdots\ W_1(\mathrm{y})\cdots\ W_2(\mathrm{x})\cdots$ (c_1 and c_2 occur)

It is important to note, that Lost Update and Write Skew phenomena are closely related to Snapshot Isolation protocol.

4.2 Snapshot Isolation protocol

Snapshot Isolation (SI) protocol was proposed in [2]. SI protocol is a special case of timetstamp based implementation of multi-version concurrency control algorithm. Under Snapshot Isolation, an unique timestamp ts_i is assigned to transaction T_i when it starts. Every write operation $W_i(\mathrm{x})$ in T_i creates a new version of data item x. New versions are not saved into database permanently (i.e. accepted) until transaction T_i is committed successfully. All read operations in T_i can only access the versions of data items either created by itself or the latest transaction that committed before T_i starts. The following example shows how the transactions under SI protocol access versions of data item.

Example 4.2.1

T_1: $R_1(x_0)$ $W_1(x_1)$ $W_1(y_1)$ C_1

T_2: $R_1(x_0)$ $W_2(x_2)$ $R_2(y_0)C_2$

T_3: $R_3(x_2)$ $R_3(y_1)$ C_3

As it is shown above, because T_1 has not committed when T_2 starts, versions of x and y created by T_1 is invisible to T_2. On the other hand, because T_1 is the committed transaction which created the latest version of y, T_3 retrieves version y_1 when it attempts to read data item y. Similarly, T_3 also reads x_2.

At this point, application of multiple versions of data items and timestamps allows for elimination of "Dirty write", "Dirty Read", "Fuzzy read", "Phantom" phenomena. However, the elimination of "Lost Update" requires an additional mechanism which is named "First-committer-wins".

"Lost Update" phenomenon is caused by the successful commitments of two transactions that performed write operations on the same data item. We say that transaction T_i is concurrent with transaction T_j when their time intervals from start-point to commit-point overlap. Transaction T_i is able to commit only when it did

not perform write operation on a data item written by another committed transaction T_j and T_j is concurrent with T_i. Otherwise, transaction T_i has to be aborted and resubmitted later. Look back at the page model of "Lost Update" anomaly, $R_1(x) \cdots W_2(x) \cdots W_1(x) \cdots c_1$. By enforcing a principle "first-committer-wins", a version of data item x created by W_2(x) must be discarded when transaction T_2 is about to commit. This is why "Lost Update" phenomenon never happens in a schedule controlled by SI protocol.

Definition 4.2.1 *Transaction T_i running under Snapshot Isolation conceptually reads data from the committed state of the database at the start time of T_i. The result of T_i's writes will be stored as a new version of data item in memory. If T_i reads data it has written, it will read the version created by itself. Snapshot Isolation also must obey a "First Committer Wins" rule, defined below.*

Definition 4.2.2 *Transaction T_i will successfully commit if and only if no concurrent transaction T_j has already committed writes of data items that T_i intends to write.*

On summary, SI is a multiversion algorithm which never delays the conflicting operations. Access to the versions of data items is controlled through managing the unique timestamp of each transaction. "First-committer-wins" guarantees that Lost update anomaly can be precluded by Snapshot Isolation.

Although high system concurrency and absence of most anomalies can be provided by Snapshot Isolation, the serializability of schedule under Snapshot Isolation can not be guaranteed all the time. In [2], a concurrency problem that can not be prevented by Snapshot Isolation was pointed out. The page model of this anomaly has been presented as "Write Skew" in section 4.1. A intuitive example of "write skew" is given below:

Example 4.2.2

Consider a combined account, which consists of a check account x and a cash account y. Each one of sub-accounts can be overdrawn as long as their total balance is not negative. Suppose the following two transactions are running under Snapshot Isolation:

T_1: $R_1(x_0 = 100)$ $R_1(y_0 = 100)$ $W_1(x_1 = -100)$ C_1

T_1: $R_2(x_0 = 100)$ $R_2(y_0 = 100)$ $W_2(y_2 = -100)$ C_2

Since the write operations are performed on different data items, no transaction in the

execution above violates the first-committer-wins principle. However, this execution sets the balance of the combined account to -200, which violates the constraint on the combined account. This defect makes conflict serializability of Snapshot Isolation not guaranteed in every possible case.

As it was stated in chapter 1, a transaction must take a database from one consistent state to another. Constraints which have been put on database should be satisfied all the time. In the example above, the constraint was satisfied initially but it was violated after the execution of schedule under Snapshot Isolation.

Recently, another defect of Snapshot Isolation was presented in [11]. This defect is named "Read-Only Transaction Anomaly". A schedule under Snapshot Isolation could be non-serializable while the execution of all update transactions is serializable. This "Read-Only Transaction Anomaly" under Snapshot Isolation is presented in the following example:

Example 4.2.3

Suppose x and y are combined accounts. Withdrawal on anyone of them will be approved by the system as long as $x + y > 0$. *However, if x+y goes negative, a penalty charge of 1 will be applied.*

T_1: $R_1(y_0, 0)\ W_1(y_1, 20)\ C_1$

T_2: $R_2(x_0, 0)\ R_2(y_0, 0)$ $W_2(x_2, -11)\ C_2$

T_3: $R_3(x_0, 0)\ R_3(y_1, 20)\ C_3$

Transaction T_2 *withdraw 10 from account x,which makes* $x+y < 0$. *Therefore, a penalty charge of 1 is applied on x. That is why the final balance of x is -11. a read-only transaction that retrieves the values of x and y and prints them out for the customer. The anomaly that arises in this transaction is that read-only transaction* T_3 *prints out* $x = 0$ *and* $y = 20$, *while the actual final values are* $y = 20$ *and* $x = -11$. *More theoretically, the result of this schedule is not equivalent to any serial schedule.*

Then, by detecting and prohibiting those non-serializable schedules, the consistency of database updated by the transactions running under SI protocol is guaranteed. A mechanism that can fulfill this objective is explained in the next section.

4.3 Characterize the serializability of Snapshot Isolation

Snapshot Isolation is an efficient multiversion concurrency control algorithm. However, it can not guarantee the serializability of concurrently execution of transactions. Recently, [10] presented an approach that can characterize the serializability of Snapshot Isolation at system design time.

The approach is based on the maintenance of a binary directed graph, Interference Graph. Interference Graph represents interferences among the transactions. Nodes in Interference Graph are committed transactions. Edges between nodes are added by observing the following rules:

- No Edge: There is no interference edge from T_j to T_k when all the following occur: $readset(T_j) \cap writeset(T_k) = \emptyset$; $writeset(T_j) \cap readset(T_k) = \emptyset$; $writeset(T_j) \cap writeset(T_k) = \emptyset$
- There is an exposed edge $T_j \xrightarrow{expo} T_k$ when both $readset(T_j) \cap writeset(T_k) \neq \emptyset$ and also $writeset(T_j) \cap writeset(T_k) = \emptyset$.
- There is an protected edge $T_j \xrightarrow{prot} T_k$ when one of two situations is found:
  - $writeset(T_j) \cap writeset(T_k) \neq \emptyset$ or
  - $readset(T_j) \cap writeset(T_k) = \emptyset$ and $writeset(T_j) \cap readset(Tk) \neq \emptyset$

When there is no need to distinguish varieties of edges, a notation(global edge) $T_j \xrightarrow{glob} T_k$ is used to indicate some interference edge exists. What should be noted here is, transactions are considered to be accessing the same data item even if operations belong to them are on different versions of it. Then a node T_k can be characterized as a "pivot" if there are transactions T_i and T_j (which may be equal to each other),such that the following three conditions all hold:

- $T_i \xrightarrow{expo} T_k$
- $T_k \xrightarrow{expo} T_j$
- T_i, T_k and T_j occur consecutively in a chord-free cycle

  $$T_a \xrightarrow{glob} T_b \xrightarrow{glob} \cdots \xrightarrow{glob} T_a$$

  in Interference Graph.

In a special case, if T_i is equal to T_j, they will all be characterized as pivots. Then, the main theorem in [10] is stated as the following.

Theorem 4.3.1 *For a set of transactions T, suppose a subset of T S is allocated Snapshot Isolation as their concurrency control mechanism. Every execution of S will be conflict-serializable if and only if none of the pivots of IG(T) is in S.*

Then, the serializability of Snapshot Isolation can be preserved by examining the corresponding Interference Graph. In the Interference Graph of a set of transactions S, if there exists pivot in IG(S), the serializability of execution of S can not be guaranteed under Snapshot Isolation. For example, the Interference Graph of the following three transactions is:
T_1: $R_1(y_0, 0)$ $W_1(y_1, 20)$ C_1
T_2: $R_2(x_0, 0)$ $R_2(y_0, 0) W_2(x_2, -11)$ C_2
T_3: $R_3(x_0, 0)$ $R_3(y_1, 20)$ C_3

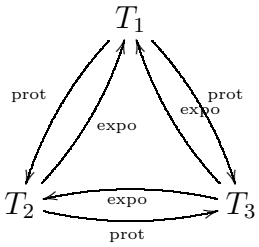


Figure 4.2: Interference Graph of Read-only transaction anomaly

Obviously, because node T_2 is in the cycle $T_3 \xrightarrow{expo} T_2 \xrightarrow{expo} T_1 \xrightarrow{prot} T_3$, T_2 is characterized as pivot. Therefore, the serializability of the concurrent execution of these transactions can not be guaranteed under Snapshot Isolation. In example 4.2.3, the schedule that is consist of these transactions is not serializable under Snapshot Isolation. In order to protect the database from corrupting, other concurrency control mechanism such as two phase locking should be allocated to Transaction T_2.

Similar with serialization graph testing that has been introduced in section 3.2.2, this "pivot" mechanism guarantees the serializability of schedule under Snapshot Isolation.

Chapter 5

Multiversion Serialization Graph for Snapshot Isolation

5.1 Motivations

According to the mechanism introduced in chapter 4([10]), transactions that may harm the serializability of schedule under Snapshot Isolation can be characterized by detecting the pivots in corresponding Interference Graph. However, this "Pivot" mechanism can only be applied at design time. The Interference Graph of all transactions that may be running concurrently will be tested. Transaction which stands for a pivot in Interference Graph will not be allowed to run concurrently with other transactions under Snapshot Isolation. In this chapter, a mechanism that can test the serializability of schedule under Snapshot Isolation at run time will be discussed.

Multiversion Serialization Graph(MVSG), proposed in [5], is a relative graph which can be used to characterize the serializability of schedule running under general multiversion concurrency control algorithm. In order to characterize the serializability of Snapshot Isolation at run time, the implementation of MVSG on schedule under Snapshot Isolation will be discussed in section 5.2. After that, a dynamic managed serialization graph especially for Snapshot Isolation is presented in section 5.3. At last, Section 5.4 will evaluate the time complexity of my proposal.

5.2 Multiversion Serialization Graph

Snapshot Isolation is an instance of multiversion concurrency control together with extra restriction "first-committer-wins". As described in chapter 3, multiversion serialization graph can be used to characterize the one copy serializability of general multiversion concurrency control. logically, it should also be applicable to Snapshot

Isolation.

Theorem 5.2.1 *A schedule S under Snapshot Isolation is one-copy-serializable if and only if the multiversion serialization graph (MVSG) is acyclic.*

For example, suppose schedule $S5.2.1$ is executed under a general multiversion concurrency control:

Example 5.2.1

Schedule $s5.2.1$

$T_1: \quad R_1(z_0) \qquad W_1(x_1) \qquad R_1(y_0) \qquad W_1(y_1) \cdots C_1$

$T_2: \qquad R_2(y_0) \qquad W_2(x_2)C_2$

$T_3: \qquad\qquad\qquad\qquad R_3(x_2) \cdots C_3$

For $W_2(x_2)$ *and* $R_3(x_2)$*, because they operate on the same versions of the same data item and* $W2(x_2)$ *precedes* $R_3(x_2)$*, there is an edge from* T_2 *to* T_3*.*
For $R_2(y_0)$ *and* $W_1(y_1)$*, because* $y_0 \ll y1$*, there is an edge from* T_2 *to* T_1*.*
For $R_3(x_2)$ *and* $W_1(x_1)$*, because* $x_1 \ll x2$*, there is an edge from* T_1 *to* T_2*.*
Consequently, the multiversion serialization graph of this schedule MVSG($S5.2.1$,$\ll$) is generated as figure 5.1.

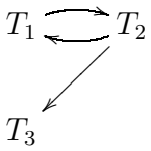


Figure 5.1: MVSG for $S5.2.1$ under a general Multiversion concurrency control

According to definition 5.2.1, cycle between T_1 and T_2 indicates the non-serializability of S.

On the other hand, if $s5.2.1$ is executed under Snapshot Isolation, transaction T_1 will be aborted when it is about to commit. To different implementation of SI, the behavior of schedule might be different. For example, in Oracle, the transaction writes the version but not commit the version will "win"(i.e. T_2 will be aborted when T_1 is committed). In this thesis, I will focus on the original conceptual of Snapshot Isolation. Two concurrent write operations on the same data item is prevented from being committed by applying "first-committer-wins". The final state of $S5.2.1$ becomes:

Schedule $s5.2.2$

T_2 : $R_2(y_0)$ $W_2(x_2)C_2$

T_3 : $R_3(x_2)\cdots C_3$

Obviously, schedule $s5.2.2$ is the subset of schedule $s5.2.1$, Consequently,MVSG($s5.2.2$,$\ll$) under Snapshot Isolation is also the subgraph of MVSG(5.2.1,$\ll$) under general multiversion concurrency control:

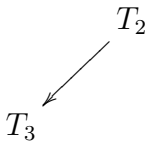


Figure 5.2: MVSG of $S5.2.2$ under Snapshot Isolation

Because transaction T_1 is aborted, and only node for corresponding committed transaction can be included in MVSG, node T_1 is excluded from graph. As the result, MVSG($s5.2.2$,$\ll$) is acyclic. So $S5.2.2$ is serializable under Snapshot Isolation.

5.3 Dynamic Management of MVSG

The serializability of schedule running under Snapshot Isolation can be guaranteed by preventing the forming of cycle in MVSG. However, because MVSG only contains nodes for all committed transactions and for no others, an assumption that significantly weakens this solution is that all the operations of concurrent transactions should be known before the graph is formed. It means that we can only apply this mechanism after the transaction's commit point. Suppose s is the set of all transactions running in a database system. Because it is not practical to predict that which transactions in s will be executed concurrently at runtime, only the latter option could be considered. In order to prevent the non-serializable execution, transactions which close a cycle in MVSG have to be aborted when they reach their commit points. However, if most concurrent running transactions have long duration life, the cost of aborting long transactions at commit point is not acceptable. Suppose two concurrent long transactions T_1 and T_2 are running concurrently under Snapshot Isolation:

At point t on the time line, the information about the transactions has been adequate to detect the non-serializability of schedule (i.e. a cycle is formed in MVSG).

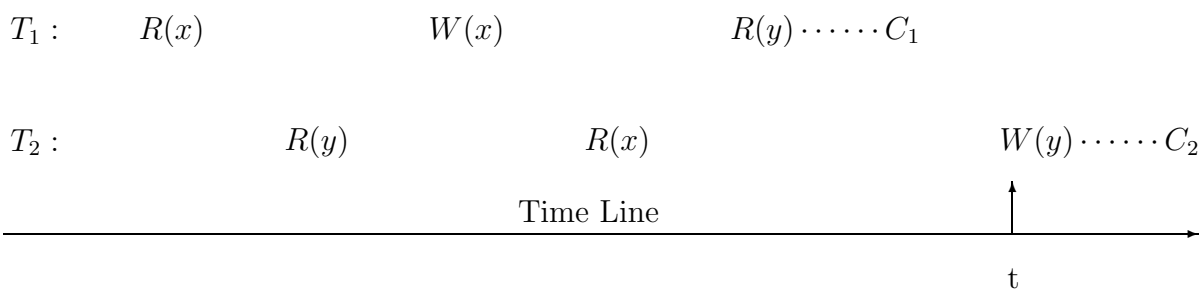


Figure 5.3: non-serializability can be found earlier

Since transaction T_2 is destined to be aborted at commit point in order to avoid the non-serializable schedule, time and system resources expended on operations of T_2 after point t are wasted. If a non-serializable execution of a long transaction can be detected at the early stages of its execution, time and system resources do not need to be wasted to continue the execution. Therefore, it is important to detect non-serializable execution as early as it is possible.

5.4 Dynamic managed Snapshot Isolation serialization graph

The non-serializability of schedule under Snapshot Isolation can be characterized by the cyclicity of corresponding MVSG. In order to avoid the waste of system resource and time when it comes long transactions, I propose that the scheduler should be aware of cycle in MVSG as soon as the operation, which may trigger an edge to close a cycle is received by scheduler. Consequently, MVSG has to be managed dynamic.

Because the original definition of MVSG is "a **serialization graph** with additional edges added", no node can be included in the graph until the corresponding transaction is committed. In order to manage the MVSG while transactions is still on going, a concept of dynamic managed serialization graph should be defined at first.

Definition 5.4.1 *Let s be a on-growing schedule, dynamic managed serialization graph (DSG) for s, denoted DSG(s), is a directed graph whose nodes are the transactions in s. Edges in DSG are all* $T_i \rightarrow T_j (i \neq j)$ *such that one of a transaction* T_i*'s operations precedes and conflicts with one of the other transaction* T_j*'s operation in s.*

A node for T_i in its DSG is added when scheduler receives the first operation of transaction T_i. As soon as an operation O_i(x) is received, possible edges between nodes in DSG will be evaluated by checking conflicting operations.

Then, for the dynamic management, the definition of MVSG can be variated as "a dynamic managed serialization graph with additional edges added". Suppose after the emergence of an edge between two write operations on different versions of the same data item, a cycle is closed in the dynamic managed MVSG(s,≪). According to Lemma 5.2.1, the corresponding schedule is supposed to be characterized as non-serializable under Snapshot Isolation. However, one of the transactions connected by that write-write edge will be aborted by "first-committer-wins". The cycle will be disappeared later. A new schedule consists of the rest transactions is serializable under Snapshot Isolation. Actually, although not all transactions in schedule s can survive at the end of its execution under Snapshot Isolation, the content of database system is consistent. Therefore, we can say schedule s is still serializable under Snapshot Isolation, to some extent. Consequently, if two concurrent transactions create versions of a same data item, the inclusion of write-write edge between these two transactions is unnecessary in a dynamic managed MVSG. As discussed in section 5.2, this write-write edge (no matter they are concurrent or not) is decided by rule 1 of definition 5.2.1. So, it can be excluded from graph if rule 1 is modified.

After the definition of dynamic managed serialization graph and the exclusion of unnecessary concurrent write-write conflict edge, a dynamic managed serialization graph especially for schedule under Snapshot Isolation can be defined as the following:

Definition 5.4.2 *For a given schedule S running under Snapshot Isolation and a version order ≪, the dynamic managed Snapshot Isolation serialization graph for S and ≪, DSISG(S,≪), is dynamic managed serialization graph(S) with the following version order edges added: for each $R_k[x_j]$ and $W_i[x_i]$ where i, j, and k are distinct,*

1. *if $x_i \ll x_j$ and T_i is not concurrent with T_j, then include $T_i \rightarrow T_j$;*
2. *if $x_j \ll x_i$, include $T_k \rightarrow T_i$.*

Conclusively, the serializability of schedule under Snapshot Isolation can be efficiently characterized by applying the following theorem:

Theorem 5.4.1 *The execution of a schedule S running under Snapshot Isolation is serializable if and only if DSISG(S,≪) is acyclic.*

Proof:
Suppose schedule s is running under Snapshot Isolation.
Firstly, assume that a DSISG(S, ≪) is acyclic. According to the definition, MVSG(S, ≪) is a DSISG(S, ≪) plus possible write-write edges between two concurrent transactions.That is, MVSG(S, ≪) is equivalent to DSISG(S, ≪) or is the superset of DSISG(S, ≪). If MVSG(S, ≪) equals to DSISG(S, ≪), MVSG(S, ≪) is also acyclic. According to Theorem 5.2.1, S is serializable. On the other hand, if MVSG(S, ≪) is the DSISG(S, ≪) plus edges between two concurrent transactions which write the same data item, one of these transactions will be forced aborted by "first-committer-wins". The execution of remain transaction is still serializable under Snapshot Isolation. Therefore, if DSISG(S, ≪) is acyclic then S is serializable.
Secondly,assume that a schedule S is serializable. Since S is serializable and Snapshot Isolation is a multiversion concurrency control algorithm, according to Lemma 5.2.1, MVSG(S, ≪) must be acyclic. According to the definition, DSISG(S, ≪) is the MVSG(S, ≪) without possible edges between two concurrent transactions which write the same data item (i.e. DSISG(S, ≪) is the subset of MVSG(S, ≪)). The subset of an acyclic graph is also an acyclic graph. So, DSISG(S, ≪) is acyclic. Therefore, if S is serializable then DSISG(S, ≪) is acyclic.

Since DSISG is a dynamic managed graph,a significant practical consideration is when the scheduler may discard the information that has been collected about a transaction. That is, remove the node for a particular transaction from the graph. To detect conflicts, the read set and write set of every transaction exists in DSISG have to be maintained. A lot of storing space will be used by this maintaining. Therefore, it is important to discard information that is not needed anymore as soon as possible. For an aborted transaction, all the information of it has been discarded automatically, so its corresponding node can be removed as soon as the transaction is aborted. For a committed transaction, one may also assume that the schedule can delete information about a transaction and remove the node as soon as it commits. Unfortunately, this is not true. For instance, consider the scheduler in example 5.2.1, if we remove the node for transaction T_2 after point C_2, the edge from T_2 to T_1 and T_2 to T_3 will be missed. Then the non-serializability which could have lead to the aborting of T_1 is missed. So, the scheduler can delete information about a committed transaction T_i if and only if T_i could not, at any time in the future, be involved in a cycle of DSISG. For a node A to form a cycle with an acyclic graph G, it must have at least one incoming edge issued

from G and one outgoing edge points to G. According to the definition of DSISG, if T_i is concurrent with T_j, edges between them can be any directions. On the other hand, if T_i is committed before T_j starts, only edge from T_i to T_j is possible. Therefore, suppose S is a set transactions and $DSISG(s, \ll)$ is acyclic. If transaction T_i is started after all transactions in S have been committed, cycle will never be formed between node T_i and S. Then the information of all transactions in S can be discarded by scheduler. The corresponding nodes for them can be removed from DSISG. In conclusion, the node of a committed transaction can be removed from DSISG if all the other transactions, which have node in DSISG, have already been committed. The following is an example of dynamic management of DSISG.

Example 5.4.1

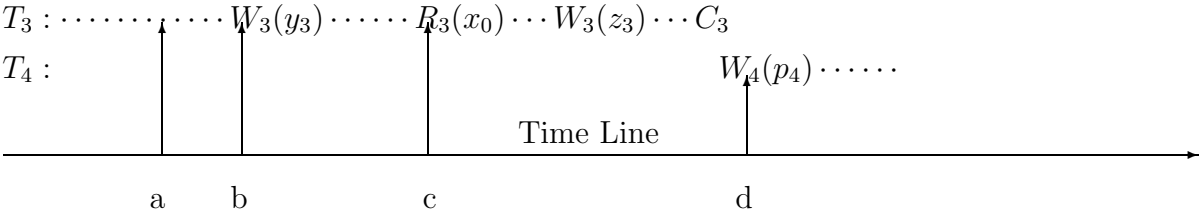
*The following is the schedule S*5.4.1 *of four executing long transactions under Snapshot Isolation*

$T_1 : R_1(x_0) \cdots C_1$

$T_2 : \cdots\cdots W_2(x_2) \cdots\cdots R_2(y_0) \cdots\cdots C_2$

$T_3 : \cdots\cdots\cdots\cdots W_3(y_3) \cdots\cdots R_3(x_0) \cdots W_3(z_3) \cdots C_3$

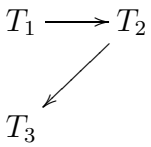
$T_4 :$ $W_4(p_4) \cdots\cdots$



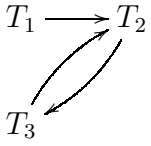
*At time a, DSISGS*5.4.1,$\ll$ *is:*

$T_1 \longrightarrow T_2$

*At time b, DSISGS*5.4.1,$\ll$ *is:*



*At time c, DSISGS*5.4.1,$\ll$ *is:*



A cycle is formed, so T_3 *is forced to abort and DSISG(S*5.4.1, $\ll$*) becomes like:*

$T_1 \longrightarrow T_2$

*At time d, DSISGS*5.4.1,$\ll$ *is:*

T_4

The serializability of $S5.4.1$ is guaranteed and DSISG($S5.4.1$, $\ll$) will keep being updated and verified as furthering operations of transactions coming.

As a conclusion, the DSISG of schedule can be created when the first operation is issued instead of at the very end of the whole schedule. A node, which indicates corresponding transaction is included when the transaction starts. Every time the scheduler receives an operation, possible edges between transactions will be evaluated (i.e. edges between nodes in the graph are updated). When a new issued edge E(T_i,T_j) between node T_i and T_j is going to close a cycle in DSISG, one of corresponding transactions T_i and T_j has to be aborted. Because the time that has been spent on a transaction can be recorded, the scheduler will choose to abort the transaction which has been run for a shorter period. This mechanism helps to save long transactions. If transaction T_i is aborted, the corresponding node is removed from graph. All nodes in DSISG can be discarded when corresponding transactions in scheduler are all committed at the moment. Consequently, system resources which used to be wasted on part of long transaction which was predetermined to be aborted can be saved.

5.5 The evaluation of time complexity

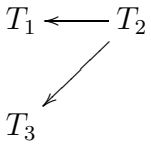
The most reasonable and recognized criteria to evaluate the efficiency of an algorithm, is the worst time complexity. To characterize an non-serializable schedule under Snapshot Isolation, the cyclicity of DSISG need to be verified. In [21], the author states that only directed acyclic graphs can be topological sorted. So, the cyclicity of DSISG or Interference Graph can be verified by topological sorting.

The worst time complexity of topological sorting a directed graph G(n,e),is O(n+e) in which n indicates the number of nodes and e stands for the number of edges. For example, suppose the schedule $S5.2.1$ in example 5.2.1 is running under Snapshot Isolation. DSISG($S5.2.1$,$\ll$) can be find in the figure 5.4.

When the topological sorting is finished, schedule can be characterized as non-serializable as soon as the sorting for DSISG is failed(i.e. there exist cycles in the DSISG).

In conclusion, for a schedule consists of n transactions, the worst time complexity of DSISG mechanism can be described as

$$T_{\text{DSISG}} = T(\text{topological sorting(DSISG)}) = T(n+e) = O(n + C_n^2) = O(n^2)$$



DSISG(S5.2.1,$\ll$)

Figure 5.4: Dynamic Managed Snapshot Isolation Serialization Graph

There is one thing should be noted here is: since DSISG is an ongoing algorithm, the time complexity discussed in this section is that of a single invocation that checks an added edge or node.

Chapter 6

Segmented Transaction Model

In order to preserve the serializability of schedule S under Snapshot Isolation protocol, a transaction whose request to access data closes a cycle in SIMVSG(S,$\ll$) should be be detected and aborted. An implementation of this mechanism is not efficient if we still apply a standard model of database transactions considered earlier. In a standard model the read and write operations of a database transaction are either organized arbitrarily or all write operations follow all read operations([10]). In the following, the necessity of a new model of transaction will be discussed from two aspects.

We need a new model because the dynamic verification of DSISG becomes inefficient when transactions are modeled in old ways. Suppose a particular write operation W is the only operation that triggers a cycle in DSISG(s). Then, non-serializability of schedule s cannot be detected until W is submitted. In the worst case, W may be delayed by the other operations to the end of the long transaction. Then, the dynamic verification of DSISG becomes no different with the previous mechanism that verify the acyclicity once at the end of transaction. For example,

Example 6.1.1
*Schedule s*6.1.1
T_1 : $R_1(x_0)$ $R_1(y_0)$ $W_1(x_1) \cdots C_1$
T_2 : $R_2(y_0)$ $R_2(x_0) \cdots$ *other operations* $\cdots W_2(y_2)C_2$
An operation $W_2(y_2)$ *issued by a transaction* T_2*, causes non-serializable execution of s*6.1.1*. If a cycle caused by* $W_2(y_2)$ *is detected earlier then it impossible to waste less time on the execution of* T_2*.*

Logically, before a change on a database can be made by a write operation, some data should be acquired or verified by a transaction.Therefore, some read operations have to be executed before a write operation. Assume that in the example above all

the preconditions of $W_2(y_2)$ are satisfied by the execution of read operations $R_2(y_0)$ and $R_2(x_0)$. Then $W_2(y_2)$ could have been submitted right after $R_2(x_0)$ in T_2. In such a case T_2 is aborted earlier without the unnecessary execution of the remaining operations of T_2.

The dynamic verification of DSISG requires identifications of the points at which the verification can be performed. Too dense or to scattered verification points make the dynamic verification either too demanding or meaningless. A concept of "breaking point" is proposed in [16], to partition a transaction into the sets of consecutive steps. An algorithm which finds the finest chopping of a set of transaction is given in [20]. This work introduces a new model of transaction, which achieves appropriate granularity without explicit "breaking points" or "transaction chopping". More efficient verification DSISG is achieved in the model through the self-revealed dependency relationships between the operations on a database.The model assumes that transaction is a sequence of operations that ends with either *commit* or *abort*. For the sake of simplicity, we only consider the transactions that end with *commit* operation. The other operations are *read* or *write*. *Write* is an operation, which signs the new value of data item that already exists in a database. *Read* retrieves a value of data item from a database. The purpose of reading the value of a data item from a database is either to inform a user about its value, or to use a value in the computations, or to verify of the logical consistency constraints imposed on a database. A *write* operation on data item x is denoted by W(x) and it is a pair $\langle (x), s \rangle$ where s is the set of data items which's values are necessary to perform W(x). We say that *write* operation on x "depends on" a data set s.

A database transaction can be logically partitioned into the segments such that each segment is concluded with a sequence of *write* operations. Moreover, the model assumes that implementation of each transactions follows a rule saying that "before write(x) is performed, a transaction reads only the data items that write(x) depends on and no other data items". So, all read operations, which access the data for user display, must be grouped between the last write operation and commit point of a transaction. Or, to those opthese The following example provides more intuitions.

Example 6.1.2

An enrollment transaction of a university administration system verifies the following consistency constraints. In this transaction, the admission offer(o) and tuition fee payment(t) should be checked. If there is no unsatisfied condition in the offer and no outstanding balance in payment, the status of student (s) will be changed to "enrolled". Also, preferred contact method (c) provided by student before should be replaced by university email account generated by system automatically. The number of student in certain school (n) will be increased by 1. At last, a welcome letter (l) retrieved from database will be print out.
Formerly, this transaction might be organized arbitrarily as:
T: R(o) R(t) R(n) W(s) R(l) W(c) W(n) C
Just as all transaction models listed in [3]. However, by following the programming rule that we proposed above, the model of this enrollment transaction will be better organized like:
T: R(o) R(t) W(s) W(c) R(n) W(n) R(l) C
In this model, transaction is logically partitioned into smaller granularities and dependencies between write operations and read operations are self-revealed.

Definition 6.1 *A segment s is a sequence of read operations followed by a write operation or commit. A segment starts either at the beginning of transaction or after a write operation and ends after the next write or commit.*

Definition 6.2 *The segmented model of database transaction is a sequence* $s_1, \cdots, s_n$, *c or where each* s_i *is a segment, c is a commit operation.*

Moreover, two additional rules that enhance standard page model(introduced in section 2.2.2) will also be applied to this segmented model. With segmented transaction model, the dynamically management and verification of DSISG can be performed at the end of each segment. When a transaction is aborted for closing a cycle in graph, the structure of this segmented model guarantees that system resource will only be wasted on minimal number of operations.

This proposal of segmented transaction model provides a tradeoff between coding freedom and system performance. Although the effectivity of DSISG mechanism will not be different whether this segmented transaction model is taken by programmer or not, the efficiency of characterizing a non-serializable transaction under Snapshot Isolation will be higher if transactions are programmed by following segmented transaction model.

Chapter 7

Self-adjusting Acyclic Serialization Graph

7.1 Motivations

DSISG is a directed graph, which presents the conflicts between concurrent transactions running under Snapshot Isolation protocol. A schedule is conflict serializable if and only if corresponding DSISG is acyclic. Unfortunately, the detection of cycle in DSISG has squared time complexity. As discussed in section 6.4, for a DSISG with n nodes and e edges, the time complexity of characterizing the acyclicity is $T = O(n+e)$. In a directed graph with n nodes, the maximum number of edges can be computed as $e = n(n-1)$. So, T can be simplified as $T = (n + n(n-1)) = n^2$. Since the number of nodes in DSISG equals to the number of concurrently running transactions, the size of graph will also increase when the number of concurrently running transactions increases . As the result, the overhead of characterizing acyclicity of DSISG grows unacceptable.

As stated in [6], a node can be involved in a cycle only if it has incoming edges and outgoing edge. Consequently, the acyclicity of DSISG can be guaranteed when: no node in the graph has both incoming and outgoing edges. Suppose the execution of operation O_i causes an edge E(T_i,T_j) between nodes T_i and T_j. If E(T_i,T_j) is approved, O_i can be executed, otherwise, transaction T_i will be aborted to eliminate the probability of forming cycle. Although it does ensure the serializability of schedule, this solution is over restricted. Suppose the probability space concers a directed edge point to a node T_i, the outcomes are {T_i has outgoing edge} and {T_i has no outgoing edge}. Because each of those two outcomes has equal chance and is independent with each other, The events of this probability space are: {T_i has outgoing edge}, with probability 0.5; {T_i has no outgoing edge}, with probability 0.5; {}=$\emptyset$, with probability 0; {T_i has outgoing edge} or {T_i has no outgoing edge}, with probability 1. Consequently,

when an incoming edge to node T_i is issued, the probability of node T_i having outgoing edges is 0.5. This is also the probability of aborting transaction T_i. This means: if an operation of transaction T_i is conflict with operation in another transaction T_j, there is 50 percents chance that T_i will be aborted. This extraordinary high frequency of aborting badly harms the concurrency of database system. In order to achive the high level of concurrency while trying to save time used to be spent on acyclicity characterization, we need a more sophisticated mechanism.

In graph theory[13], a cycle in directed graph is defined as "A closed directed walk, with repeated nodes allowed". This means that if a node N is in a cycle, a directed walk started from N can finally reach N.

Definition 7.1.1 *In a directed graph, every nodes that can be reached by following edges started from N are called successors of N. Likewise, all nodes that can reach N by following edges started from themselves are called ancestors of N.*

acyclic directed graph can be guaranteed by observing the following rule:

Conclusion 7.1.1 *A directed graph G is acyclic, if and only if no node in G is pointed by its successor.*

Proof:
Suppose there is no node in a cyclic graph G is pointed by its successor. Since G is cyclic, there is path starts from N_i and travels through one of its successors N_j and reaches N_i again. Then N_i is pointed by its successor N_j. It's a contradiction. So, if there is no node in graph G is pointed by its successor, G is acyclic.

Suppose in acyclic graph G, there is a node N_i is pointed by its successor N_j. According to definition 7.1.1, there exists a directed path from N_i to its successor N_j. Moreover, N_i is pointed by N_j. So there exists a directed path starts from N_i, travels through N_j and reaches N_i again, which means a cycle. This is contradicted with that G is acyclic. So, if G is acyclic, there is no node can be pointed by its successor.

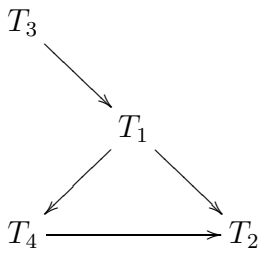
When an edge is issued from node N_i to N_j, all successors of N_j will be checked to find out whether N_i is one of them. If not, the edge from N_i to N_j can be allowed. Obviously, if the number of successors of a node is large, the time spend on going through all successors will be high. In the next section, I will present a new structure

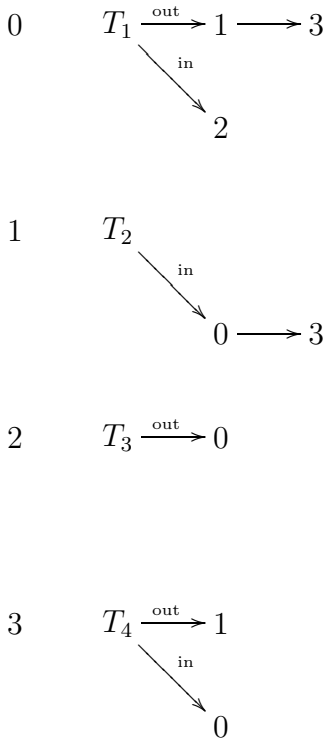
of directed graph. In that well organized directed graph, the relationship between nodes will become explicit without checking the whole set of their successors.

7.2 Self-adjusting acyclic graph

Self-adjusting acyclic graph is an acyclic directed graph. The corresponding node of a transaction will be included in graph when the transaction is started. nodes will be assigned with unique level values when they are connected by conflict edges. The level value of node N_i is indicated as LV(N_i). E(T_i,T_j) stands for an edge from node T_i to T_j.CS(N_i) is the collection of successors of node N_i. AS(N_i) is the collection of ancestors of node N_i. Before the graph is created, two variables NV=0 and PV=0 are initialized. NV and PV records the level value of top level and bottom level, respectively. The graph can be implemented by "adjacency list". "Adjacency list" is a data structure which consists of an n-elements(n is the number of all nodes in graph) array of linked lists. In position i of the array, the information of node N_i and two pointers to the linked list of edges connected with N_i are stored. One linked list List_out stores the index number(position in the array) of nodes that are connected by edges which incident from N_i. The other linked list List_in stores the index number(position in the array) of nodes that are connected by edges which point to N_i. A directed graph and corresponding Adjacency list is presented in the following example:

Example 7.2.1





When a transaction is started, the corresponding node of a transaction is included in graph. Before the node is connected with other nodes by conflict edges, its level value is initialized as zero. Obviously, if LV(N_i)=0, it means that N_i is neither ancestor nor successor of any other nodes. We call this kind of node "isolated node". To an isolated node N in acyclic graph G, the acyclicity of G will not be harmed no matter an incoming or outgoing edge is newly connected with N. Consequently, when E(T_i,T_j) is issued, if LV(T_i) or LV(T_j) equals zero, the edge can be approved immediately. If LV(T_i)=0, NV is decreased by 1 and assigned to LV(T_i). On the other hand, if LV(T_j)=0, PV is increased by 1 and assigned to LV(T_j). The level value of initial node comes from NV which's value is always decreasing. The level value of terminal node comes from PV which's value is always increasing. Consequently, it can be concluded that:

Conclusion 7.2.1 *In self-adjusting acyclic graph, the level value of an node is always smaller than level value of its successor.*

By applying rules above, the self-adjusting acyclic graph can be intuitively looked as a directed graph with numbers of levels. On each level there is only one node. The level with greater value is regarded "below" the one with smaller value. A directed edge from a node in upper level to lower level is called "pointing down". Otherwise, it is

"pointing up".

For two nodes T_i and T_j,if $LV(T_i) < LV(T_j)$, there is no possibility that T_i is the successor of T_j. The edge which is "pointing down" from T_i to T_j will not close a cycle in the graph. So, Suppose none of T_i and T_j is isolated node. When edge E(T_i,T_j) is issued, if $LV(T_i) < LV(T_j)$, E(T_i,T_j) can be approved immediately and there is no variation to $LV(T_i)$ and $LV(T_j)$.

On the other hand, if the edge is going to "pointing up", the situation becomes more complex. Because there is E(T_i, T_j) and $LV(T_i) > LV(T_j)$, according to Conclusion 7.2.1, T_i might be the successor of T_j. If T_i is the successor of T_j, the acyclicity of graph can not be preserved. Therefore, this "pointing up" edge has to be declined. If T_i is not the successor of T_j, E(T_i,T_j) will not form a cycle in graph so it can be approved. Moreover, this approval of "pointing up" edge E(T_i,T_j) will result in that T_i has a successor T_j which is assigned with a smaller level value. In order to overcome this violation of Conclusion 7.2.1, I propose that the approval of "pointing up" edge has to come together with a procedure which is named "pushing down". When the "pointing up" edge E(T_i,T_j) is approved, T_j and all its successors will be pushed down to other levels which are lower than T_i. Consequently, all approved edges in self-adjusting acyclic graph are still pointing from higher to lower level. So, when an edge is issued from a node T_i in lower level to a node in higher level T_j, the algorithm will firstly get CS(T_j). If T_i belongs to CS(T_j), edge E(T_i,T_j) is refused. Otherwise, E(T_i,T_j) can be approved and the level values of nodes in CS(T_j)) will be re-assigned.

The algorithm that manages edges in self-adjusting acyclic graph is presented below:

Algorithm 7.2.1

When E(T_i,T_j) is issued

Procedure SAAG BEGIN

```
1     IF LV(T_i) * LV(T_i) = 0 then
2         Approve_Edge(E(T_i,T_j));
3     ELSE
4         IF LV(T_i) < LV(T_j) then
5             Approve_Edge(E(T_i,T_j));
6         ELSE
```

```
7               CS SetOfnodes;
8               CS := Get_CS(T_j); *
9               IF T_i in CS THEN
10                  Decline E(T_i,T_j);
11              ELSE
12                  Approve_Edge(E(T_i,T_j));
13                  Push_Down(T_j,CS);
            End IF;
        END IF;
    End IF;
END SAAG;
```

*In this statement, CS(T_j) is obtained by calling classical graph traverse algorithm Depth-first search, DFS(T_j).

```
Procedure Approve_Edge(e Edge)
  IF LV(T_i)=0 then
      NV := NV-1;
      LV(T_i) := NV;
  End IF;
  IF LV(T_j)=0 then
      PV:=PV+1;
      LV(T_j) :=PV;
  End IF;
  Build_Edge(e);

Procedure Push_Down(N node, CS SetOfNodes) IS
  BEGIN
      CS := CS ⋃ {N}
      TPV :=PV;
      P := LV(N)
      For i:=0..Length(CS)-1 Loop
          Distance := LV(CS(i))- P;
          LV(CS(i)) := PV+Distance+1;
          IF LV(CS(i)) > TPV THEN
          TPV := LV(CS(i));
```

```
            END IF;
        End Loop;
        PV := TPV;
    End PushDown;
```

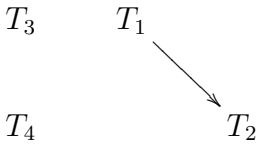
In the following example, the above algorithm will be clarified to show the variation of a self-adjusting acyclic graph.

Example 7.2.2

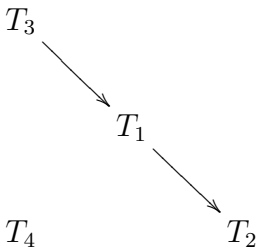
T_1 T_2

T_3 T_4

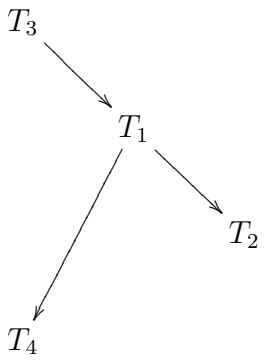
Four nodes are included in a self-adjusting acyclic graph G.
$LV(T_1) = LV(T_2) = LV(T_3) = LV(T_4) = 0$, $NV = 0$, $PV = 0$



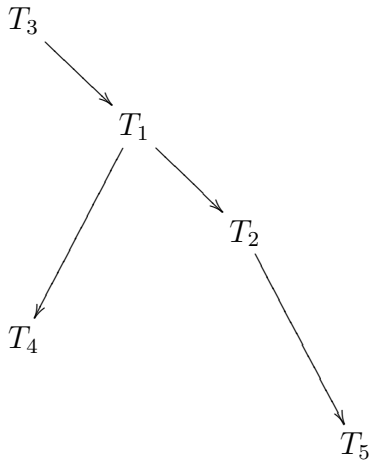
Edge E(T_1, T_2) is issued, because $LV(T_1) * LV(T_2) = 0$, the edge is approved.
$LV(T_1) = -1$, $LV(T_2) = 1$, $LV(T_3) = LV(T_4) = 0$, $NV = -1$, $PV = 1$



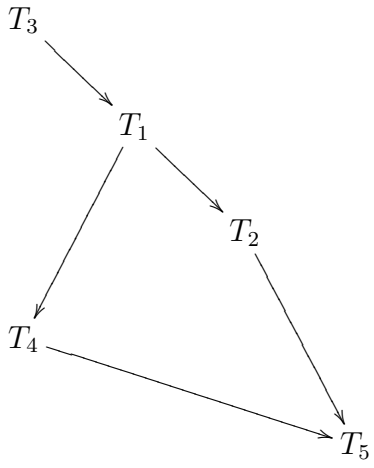
Edge E(T_3, T_1) is issued, because $LV(T_1) * LV(T_3) = 0$, the edge is approved.
$LV(T_1) = -1$, $LV(T_2) = 1$, $LV(T_3) = -2$, $LV(T_4) = 0$, $NV = -2$, $PV = 1$



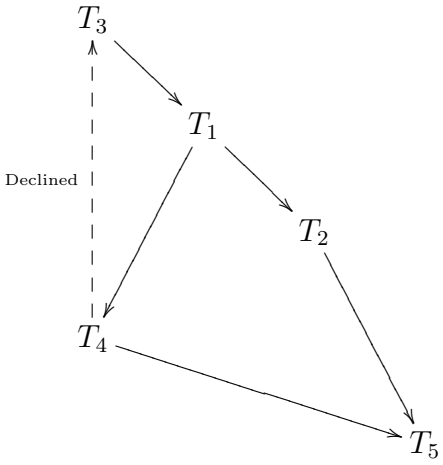
Edge E(T_1, T_4) is issued, because $LV(T_1) * LV(T_4) = 0$, the edge is approved. $LV(T_1) = -1$, $LV(T_2) = 1$, $LV(T_3) = -2$, $LV(T_4) = 2$, $NV = -2$, $PV = 2$



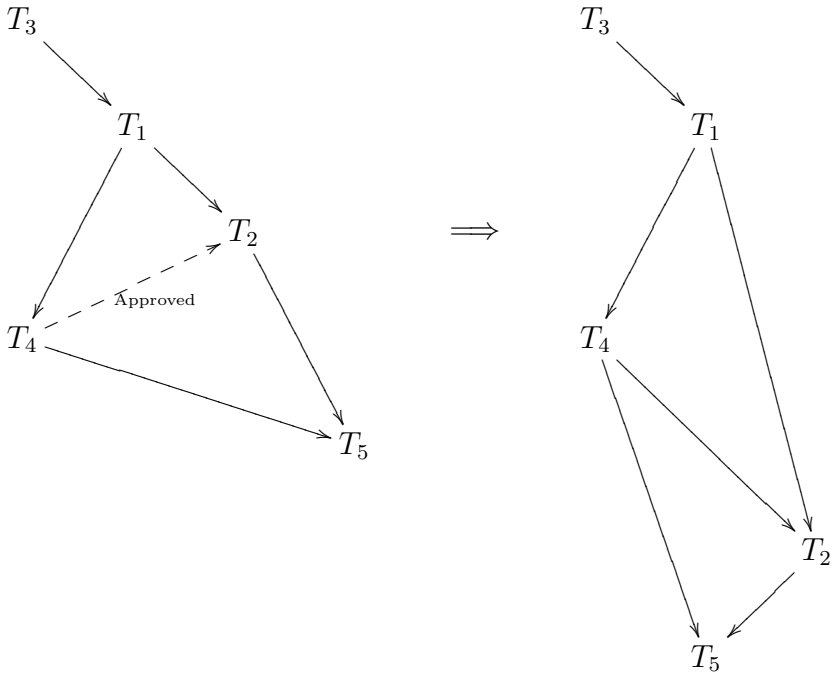
A new node T_5 is included in G, because $LV(T_2) * LV(T_5) = 0$, the edge is approved. $LV(T_1) = -1$, $LV(T_2) = 1$, $LV(T_3) = -2$, $LV(T_4) = 2$, $LV(T_5) = 3$, $NV = -2$, $PV = 3$



Edge E(T_4, T_5) is issued, because the edge is pointing from higher level to lower level, the edge is approved.
$LV(T_1) = -1$, $LV(T_2) = 1$, $LV(T_3) = -2$, $LV(T_4) = 2$, $LV(T_5) = 3$, $NV = -2$, $PV = 3$



Edge E(T_4, T_3) is issued, because the edge is pointing from lower level to higher level and $T_4 \in CS(T_3)$, the edge is declined.
$LV(T_1) = -1$, $LV(T_2) = 1$, $LV(T_3) = -2$, $LV(T_4) = 2$, $LV(T_5) = 3$, $NV = -2$, $PV = 3$



Edge E(T_4, T_2) is issued, although the edge is pointing from lower level from higher level, $T_4 \notin CS(T_2)$,so the edge is approved. T_2 and its successor T_5 are pushed to lower

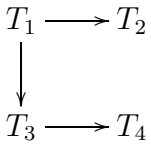
levels.
$LV(T_1) = -1$, $LV(T_2) = 4$, $LV(T_3) = -2$, $LV(T_4) = 2$, $LV(T_5) = 5$, $NV = -2$, $PV = 5$

Obviously, except the execution of Get_CS(T_i), the other parts of algorithm only have linear complexity. Get_CS(T_i) is the function that return the reachable set of a node T_i in directed graph G. Depth-First Search DFS(n) is a classical algorithm. It can be used to visit every node which is reachable to N, in a directed Graph. According to [19], the worst time complexity of DFS(n) is $O(n^2)$. Consequently, the worst time complexity of function Get_CS(T_i) is $O(n^2)$.

7.3 Parameterized Self-adjusting Acyclic Graph

Although the average time complexity of algorithm above is lower than that of traditional serialization graph testing, it is still on the squared level and they have the same worst time complexity $O(n^2)$. In order to decrease the time complexity, the rule of evaluating pointing-up edge E(T_i,T_j) in Self-adjust acyclic graph can be refined. Instead of getting the whole CS(T_j), the retrieving of CS(T_j) can be parameterized. Parameter L is introduced to limit the length of edges that can be traveled when retrieving the set of successors of node T_j. If $T_i \in$ Get_CS(T_j,L) or there exists node T_a that is reachable from T_j and the length of edges from T_j to T_a is more than L, the edge E(T_i,T_j) will be declined. The following graph gives an intuitive example of the concept of "Length of Edges".

Example 7.3.1

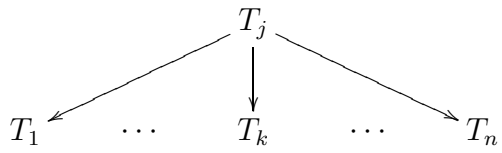


From T_1 to T_2, the length of edge that has been traveled is 1.
From T_1 to T_4, the length of edge that has been traveled is 2.

In an acyclic directed graph which has N nodes, the maximum length of edges is N-1. So,the value of L is an integer which stands in range $[1, (n-1)]$. When an edge E(T_i,T_j) is issued and $LV(T_i) * LV(T_j) \neq 0$, $LV(T_i) > LV(T_j)$, Get_CS(T_j,L) will only travel L length of edges to get the set of successors of node T_j. When L=1, the set of

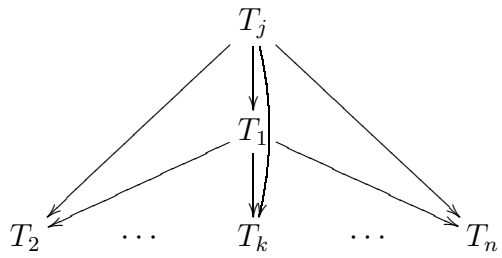
nodes that directly connected with T_j will be returned as CS(T_j). When $L = n - 1$, the whole set of CS(T_j) will be returned. Consequently, the worst time complexity of algorithm variates with the value of parameter L. When an "pointing up" edge E(T_i,T_j) is issued, Get_CS(T_j,L) will retrieve the set of successors of T_j by traveling L length of edges.

When L=1, the worst case is that all the other nodes(except T_j) are the successor of T_j and maximum length of edges is 1.



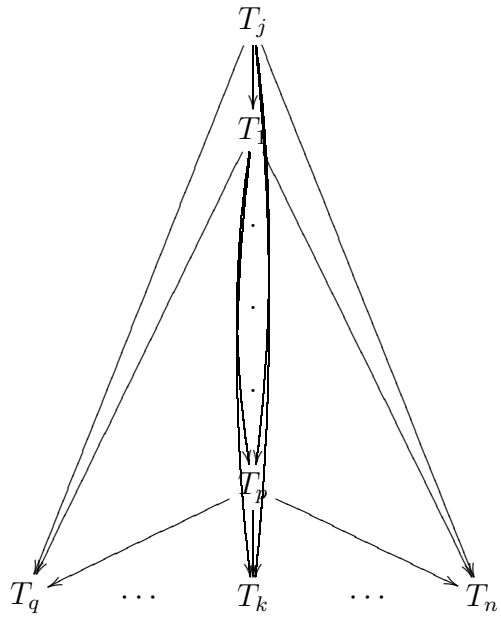
The time spent on traversing the above graph G is the time spent on traveling through each edge in G. So, the maximum time spent on Get_CS(T_j,1) is $T(1) = N - 1$.

When L=2, the worst case is that all the other nodes(except T_j) are the successor of T_j and maximum length of edges is 2.



The maximum time spent on Get_CS(T_j,2) is $T(2) = (N - 1) + (N - 2) = 2N - 3$

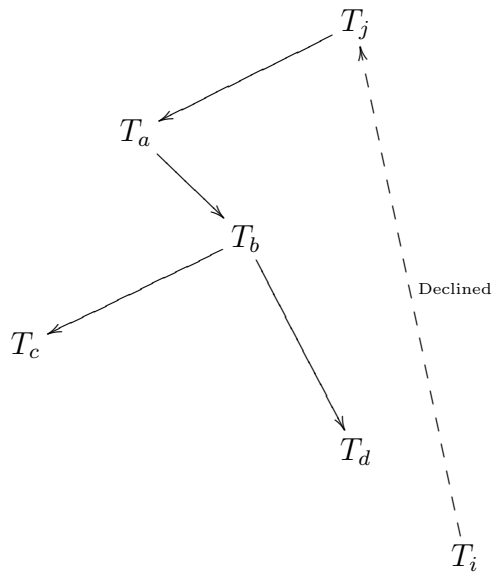
When L=K, the worst case is that all the other nodes(except T_j) are the successor of T_j and maximum length of edges is K.



The maximum time spent on Get_CS(T_j,K) is $T(K) = (N-1)+(N-2)+\cdots+(N-K) = KN-(1+K)K/2$

Obviously, when $L < N-1$, the worst time complexity of Get_CS(T_j,L) is O(N). When L=N-1, time spent on Get_CS(T_j,K) is $T(N) = N*N-(1+N)N/2 = (N^2-N)/2$. This is exactly the same as the time spent on retrieving the whole set of successors of T_j. Consequently, when L=N-1, the worst time complexity is $O(N^2)$.Although the worst time complexity can be decreased from squared to linear, a downside is introduced by parameterized algorithm. When $L < N-1$, some edge might be declined unnecessarily. For example, in the following graph, when L=2, edge E(T_i,T_j) which will not cause a cycle is declined unnecessarily by the algorithm.

Example 7.3.1



When L=2, E(T_i,T_j) is unnecessarily declined

In order to minimize the impact of this unnecessary refusion, a mechanism called "pushing up" is proposed. When a "pointing up" edge E(T_i,T_j) is issued and there exists node T_a that is reachable from T_j and the length of edges from T_j to T_a is more than L and T_i is not in the set that returned by Get_CS(T_j,L), the algorithm will call a function Get_AS(T_i,L). If $T_j \notin AS(T_i, L)$ and AS(T_i,L) is the full set of all ancestors of node T_i, E(T_i,T_j) is approved and AS(T_i,L) is pushed up to other levels together with node T_i. This "pushing up" mechanism is a complement to "pushing down" such that the chance of unnecessary refusion is decreased by half. The parameterized algorithm is presented below:

Algorithm 7.3.1

When E(T_i,T_j) is issued
CS SetOfnodes;
AS SetOfnodes;
Full_CS boolean;
Full_AS boolean;
Procedure SAAG_Par(L Integer) Begin
1 IF $LV(T_i) * LV(T_i) = 0$ then
2 Approve_Edge(E(T_i,T_j));
3 ELSE
4 IF $LV(T_i) < LV(T_j)$ then
5 Approve_Edge(E(T_i,T_j));

```
6           ELSE
7               Get_CS(T_j,L);
8               IF T_i in CS THEN
9                   Decline E(T_i,T_j);
10              ELSE
11                  IF Full_CS=true then
12                      Approve_Edge(E(T_i,T_j));
13                      Push_Down(T_j,CS);
14                  ELSE
15                      Get_AS(T_i,L);
16                      IF T_j in AS THEN
17                          Decline E(T_i,T_j);
18                      ELSE
19                          IF Full_AS=true then
20                              Approve_Edge(E(T_i,T_j));
21                              Push_Up(T_i,AS);
22                          ELSE
23                              Decline E(T_i,T_j);
24                          END IF;
25                      END IF;
26                  END IF;
27              END IF;
28          END IF;
29      End IF;
END SAAG_Par;

    Procedure Get_CS(V in node, Length in Integer) *
BEGIN
    Get the set of successors of node N without traveling more than L length;
    CS := the result of statement above;
    IF CS is the full set of successors of node N then
        Full_CS:=true;
    ELSE
        Full_CS:=false;
    END IF; END Get_CS;
```

*In this procedure, set of successors is obtained by calling classical graph traverse algorithm Depth-first search, DFS(T_j). Link list List_out in adjacency list of graph is used. If DFS(T_j) is terminated because the length of edges that can traveled exceeds L, CS is not the full set of ancestors of node N. Otherwise, CS is the full set.

```
    Procedure Get_AS(V in node, Length in Integer) *
BEGIN
     Get the set of ancestors of node N without traveling more than L length;
     AS := the result of statement above;
     IF AS is the full set of ancestors of node N then
          Full_AS:=true;
     ELSE
          Full_AS:=false;
     END IF;
END Get_CS;
```

*In this procedure, set of ancestors is also obtained by calling classical graph traverse algorithm Depth-first search, DFS(T_j). Link list List_in in adjacency list of graph is used. If DFS(T_j) is terminated because the length of edges that can traveled exceeds L, AS is not the full set of ancestors of node N. Otherwise, AS is the full set.

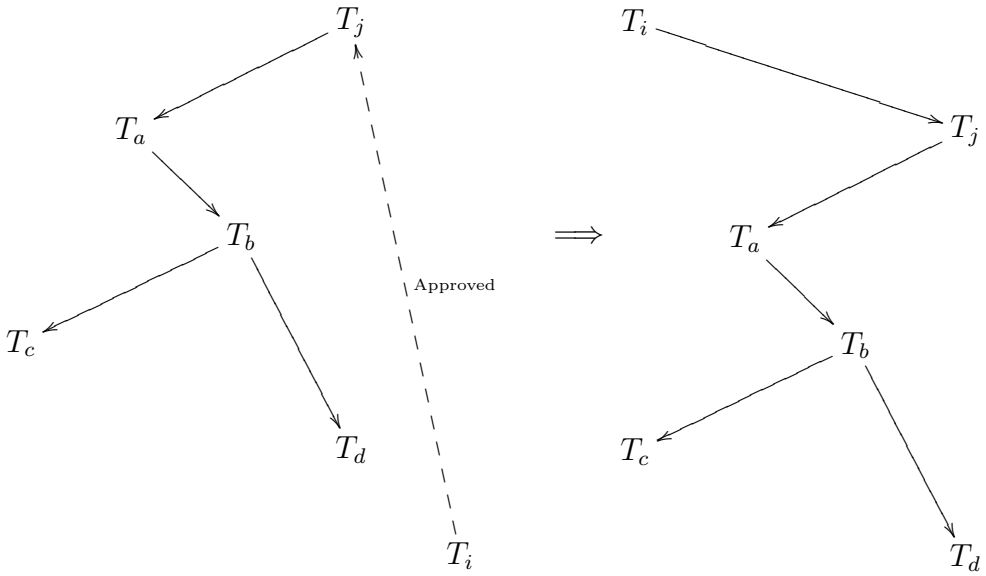
```
    Procedure Approve(E Edge)
     IF LV(T_i)=0 then
          NV := NV-1;
          LV(T_i) := NV;
     End IF;
     IF LV(T_j)=0 then
          PV:=PV+1;
          LV(T_j) :=PV;
     End IF;
     Build_Edge(e);

    Procedure Push_Down(N node, CS SetOfNodes) IS
     BEGIN
          CS := CS ∩ N
          TPV :=PV;
```

```
        P := LV(N)
        For i:=0..Length(CS)-1 Loop
            Distance := LV(CS(i))- P;
            LV(CS(i)) := PV+Distance+1;
            IF LV(CS(i)) > TPV THEN
            TPV := LV(CS(i));
            END IF;
        End Loop;
        PV := TPV;
    End PushDown;

    Procedure Push_Up(N node, AS SetOfnodes) IS
     BEGIN
        AS := AS ∩ N
        TNV :=NV;
        P := LV(N)
        For i:=0..Length(AS)- 1 Loop
            Distance := LV(AS(i))- P;
            LV(AS(i)) := NV - Distance - 1;
            IF LV(AS(i)) < TNV THEN
            TNV := LV(AS(i));
            END IF;
        End Loop;
        NV := TNV;
    End PushDown;
```

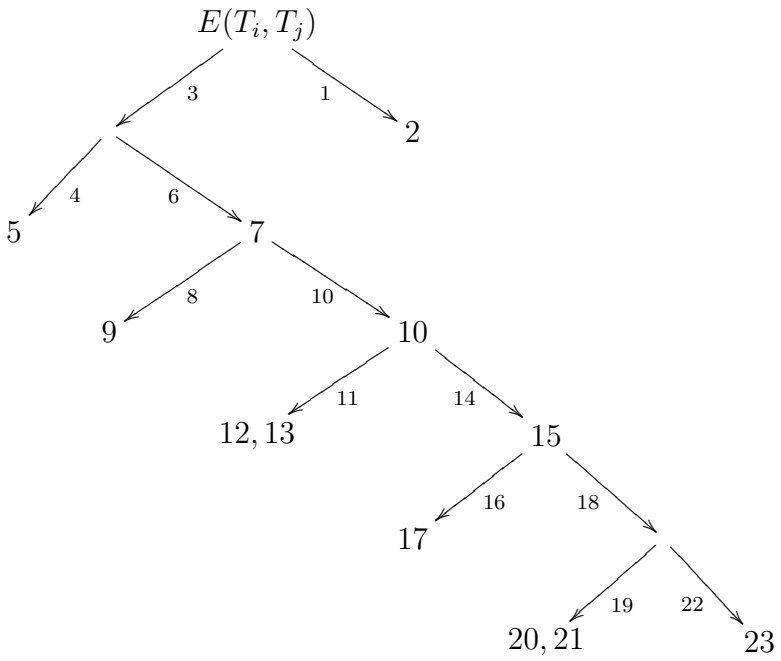
In example 7.3.1, edge E(T_i,T_j) can be "saved" by mechanism of "pushing up". Because Get_CS(T_j,L) does not return the whole set of CS(T_j) and $T_j \notin CS(T_j, L)$, the algorithm will check the set returned by Get_AS(T_i,L). AS(T_i,L) is the whole set of ancestors of T_i and $T_i \notin AS(T_i, L)$. So E(T_i,T_j) is approved and T_i is pushed up.



E(T_i,T_j) in example 7.3.1 *is saved by "pushing up" mechanism*

The parameterized algorithm provides a tradeoff between efficiency and accuracy. The value of L can be adjusted based on actual situation. If the number of concurrently running transactions is small, which is the same as number of nodes in self-adjusting acyclic graph, the overhead caused by squared time complexity will not give pressure to the system. L can be set to N-1. On the other hand, if the number of concurrently running transaction can be potentially high, the efficiency is a bigger concern than the accuracy of algorithm. Then L can be set as proper number which is less than N-1. Since the performance of algorithm will be enhanced significantly(from squared complexity to linear complexity), these "unnecessary" rejections are endurable sacrifice. Moreover, in the following decision tree of algorithm 7.3.1, numbers are the line numbers of pesudo-code in algorithm 7.3.1. In a probability space concerns the execution of algorithm 7.3.1, the outcomes are the execution of code which's line number is indicated by leaf nodes of decision tree. The events of this probability space are: {line 2}, with probability $(\frac{1}{2})$; {line 5}, with probability $(\frac{1}{2}^2)$; {line 9}, with probability $(\frac{1}{2}^3)$; {line 12 and line 13}, with probability $(\frac{1}{2}^4)$; {line 17}, with probability $(\frac{1}{2}^5)$; {line 20 and line 21}, with probability $(\frac{1}{2}^6)$; {line 23}, with probability $(\frac{1}{2}^6)$; {any one of code in line indicated by leaf nodes}, with probability 1. When edge E(T_i,T_j)is issued in self-adjusting acyclic graph, E(T_i,T_j) might be declined uncessarily only when statement 23 is executed. So, the probability of unnecessary rejection on this edge is only $(\frac{1}{2})^6$. This is acceptable to a morden database system which's efficiency is always the most

important concern.



7.4 Implement the SAAG on Snapshot Isolation Protocol

Because self-adjusting acyclic graph is the mechanism that efficiently preserves the acyclicity of graph, it can be implemented on Snapshot Isolation protocol to preserve the serializability of schedule under SI at run-time.

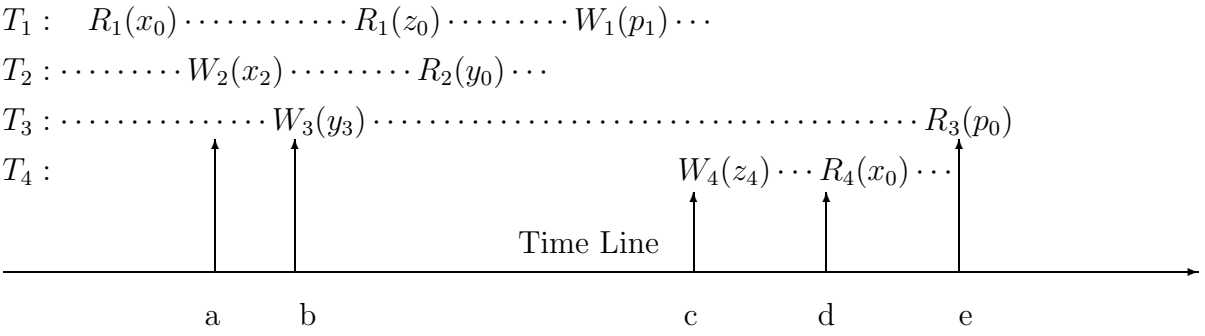
In section 6.2, dynamic managed Snapshot Isolation serialization graph was introduced. DSISG is a serialization graph that can ensure the serializability of Snapshot Isolation protocol at run-time, by preserving the acyclicty of graph. The disadvantage of DSISG is that N^2(N is the number of nodes in DSISG) time will be spent on validating the acyclicity of graph. If DSISG is structured by applying algorithm 7.3.1, the cost of having a serializable SI protocol can be decreased. With the tolerance of seldom "unnecessary aborting of transactions", the worst time complexity can even be degraded from squared to linear.

Suppose Snapshot Isolation is set as the concurrency control protocol of a database system. When a transaction is started, the corresponding node will be included in DSISG. Edges in DSISG are issuing according to definition 5.4.2. When an edge E is issued, it will be approved or declined by applying algorithm 7.3.1. If E is approved,

the transaction that issues E can be carried on. On the other hand, if E is declined, the transaction that issues E will be aborted. The corresponding node of this transaction together with all edges attached with it are removed from DSISG. To a committed transaction T, the corresponding node can be removed from DSISG if there are no active transactions which are concurrent with T.

For example, suppose parameter L is set as 2 to algorithm 7.3.1 in the database system. The execution of concurrently running transactions under Snapshot Isolation protocol is presented as below:

Example 7.4.1

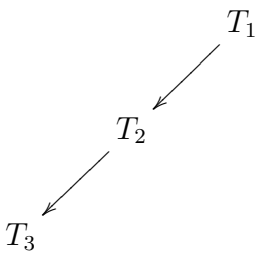


At time a, because LV(T_1)* LV(T_2)=0, DSISG is:

T_3 T_4

LV(T_1)=-1, LV(T_2)=1, LV(T_3)=0, LV(T_4)=0, NV=-1, PV=1

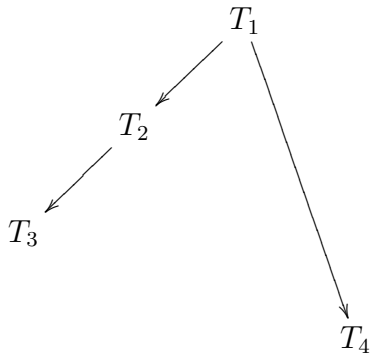
At time b, because LV(T_2)* LV(T_3)=0, DSISG is:



T_4

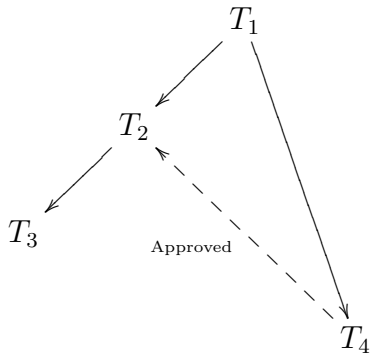
LV(T_1)=-1, LV(T_2)=1, LV(T_3)=2,LV(T_4)=0, NV=-1, PV=2

At time c, because LV(T_1)* LV(T_4)=0, TDSISG is:

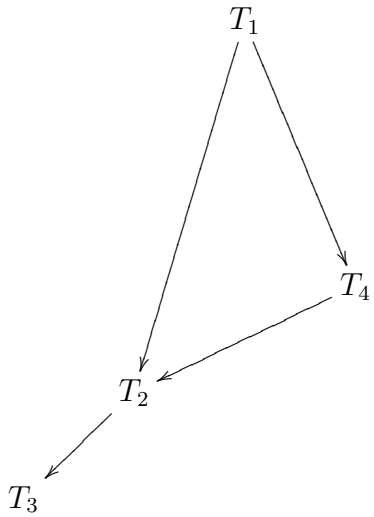


LV(T_1)=-1, LV(T_2)=1, LV(T_3)=2,LV(T_4)=4, NV=-1, PV=4

At time d, because LV(T_2)* LV(T_4) $\neq$ 0, LV(T_2)< LV(T_4), $T_4 \notin$ CS(T_2), DSISG is:

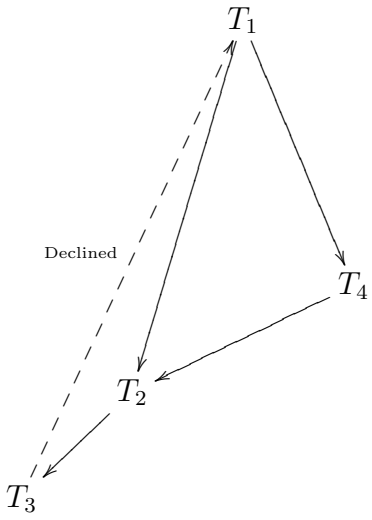


E(T_4,T_2) is declined, node T_2 and T_3 are pushed down.

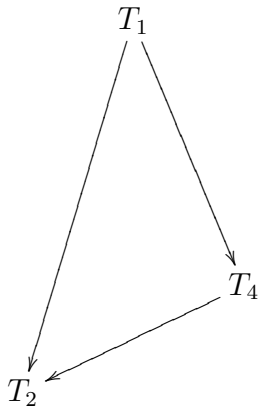


LV(T_1)=-1, LV(T_2)=5, LV(T_3)=6,LV(T_4)=4, NV=-1, PV=6

At time e, because LV(T_1)* LV(T_3) $\neq$ 0, LV(T_1)< LV(T_3), $T_3 \in$ CS(T_1), DSISG is:



E(T_3,T_1) is declined, suppose transaction T_3 has been run longer than transaction T_1, according to the mechanism introduced at the end of section 5.4,transaction T_3 is aborted. Node T_3 is removed from graph.



LV(T_1)=-1, LV(T_2)=5,LV(T_4)=4, NV=-1, PV=6

As the result, Snapshot Isolation is preserved serializable by using DSISG which is managed by algorithm 7.3.1.

Chapter 8

Contributions and Open Problems

In this thesis, I provided a new mechanism that makes Snapshot Isolation protocol(SI) serializable. This mechanism has the following properties:

1. it characterizes the non-serializable transactions under SI at run time by detecting cycles in a Dynamic Managed Snapshot Isolation Serialization Graph(DSISG);

2. it guarantees the serializability of SI by preserving DSISG acyclic without the time-consuming acyclicity validation of DSISG.

After the study of Snapshot Isolation, I proved that a variation of classical Multiversion Serialization Graph can be used to characterize the serializability of Snapshot Isolation. Transaction that will cause a cycle in the revised MVSG will be characterized as non-serializable under SI. Then, I discussed the performance aspects of this approach when it is dealing with long transaction. If a long transaction is characterized as non-serializable and aborted at the very end, the overhead of system becomes unacceptable. In order to avoid this situation, the dynamic management of graph is presented. This dynamic managed graph is named DSISG.

I found that the dynamic management of DSISG is not effective when transactions are programmed by observing the traditional models. After studying the semantic relationships between operations in transaction, a segmented model of database transaction is defined. The efficiency of DSISG is enhanced by following this segmented model.

Like all graph based concurrency control mechanisms, DSISG suffers from a square worst time complexity because of the validating of graph acyclicity. I proposed a hierarchical structure of the graph so that DSISG can be preserved acyclic without that high cost. Nodes are placed in different level in self-adjusting acyclic graph and edges

are only allowed from higher level to lower level. There is no need for my mechanism to follow the traditional process of serialization graph testing, "verifying the acyclicity → approving/rejecting". Two out of three types of issued edges(1, edge from higher level to lower level and no isolated node is involved; 2, edge involves at least one isolated node) can be approved directly. For the third type of edge(edge from lower level to higher level and no isolated node is involved), numbers of rules are used to validate it. The efficiency and precision of validation are determined by the value of a parameter "L". When $L = N - 1$, my mechanism has the same worst time complexity with the one which use topological sorting to validates the acyclicity of normal DSISG. However, my mechanism has lower average time complexity. When $L < N - 1$, the worst time complexity of my mechanism is decreased to linear, with $(\frac{1}{2})^6$ probability in which edge may be declined unnecessarily. The administrator of database system can tune the value of parameter L by observing the performance of system. Higher the value of L is, more time will be spent on managing graph and fewer transaction will be aborted unnecessarily. If squared worst time complexity is acceptable in particular database system, L can be set to N-1 to acquire the best precision of mechanism.

Conclusively, my contributions in this thesis are:

- A graph based approach which characterizes the serializability of Snapshot Isolation protocol at run time, especially when dealing with long transactions.
- A segmented model of database transaction.
- An graph constructing mechanism which preserves the acyclicity of graph more efficiently. More flexibility and efficiency can be obtained when this mechanism is applied to approach in contribution one.

The further research on this topic will go through the following steps. Firstly, the reusing of level value in acyclic graph will be studied. Secondly, an implementation of self-adjusting acyclic graph will be developed. Thirdly, the program will be implemented in a database system to work out a criteria of adjusting parameter L. Snapshot Isolation will be used as concurrency control method of this database system. Finally, the hierarchical structure of serialization graph will be generalized so that it can be applied to concurrency control algorithms other than Snapshot Isolation protocol.

Bibliography

[1] C. Beeri, P.A. Bernstein, and N. Goodman. A model for concurrency in nested transaction systems. In *Journal of the ACM 36*, pages 230–269. ACM Press, 1989.

[2] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 1–10, New York, NY, USA, 1995. ACM Press.

[3] Arthur Bernstein, Phil Lewis, and Shiyong Lu. Semantic conditions for correctness at different isolation levels. *Proceedings of the 16th International Conference on Data Engineering (ICDE'2000)*, pages 57–66, 2000.

[4] P.A. Bernstein, D.W. Shipman, and W.S. Wong. Formal aspects of serializability in database concurrency control. In *IEEE Transactions on Software Engineering SE-5*, pages 75–101. IEEE Press, 1979.

[5] Philip A. Bernstein and Nathan Goodman. Multiversion concurrency control-theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, 1983.

[6] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[7] Y. Breitbart and A. Siberschatz. Strong recoverability in multidatabase systems. In *In Proc. 2nd Inernational Workshop on Research Issues in Data Engineering-Transaction and Query Processing.*, pages 170–175. ACM Press, 1992.

[8] M.A. Casanova. The concurrency control problem for database systems. *Lecture Notes in Computer Science*, 116, 1981.

[9] K.P. Eswaran, J. Gray, R.A. Lorie, and I.L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19:624–633, 1976.

[10] Alan Fekete. Allocating isolation levels to transactions. In *PODS '05: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 206–215, New York, NY, USA, 2005. ACM Press.

[11] Alan Fekete, Elizabeth O'Neil, and Patrick O'Neil. A read-only transaction anomaly under snapshot isolation. *SIGMOD Rec.*, 33(3):12–14, 2004.

[12] Jim Gray and Andreas Reuter. *Transaction Processing : Concepts and Techniques*. Morgan Kaufmann, 1993.

[13] Jonathan L. Gross and Jay Yellen. *Handbook of Graph Theory*. Boca Raton : CRC Press, 2004.

[14] C.A.R Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12, 1969.

[15] H. K. Korth and G. Speegle. Formal model of correctness without serializabilty. In *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 379–386, New York, NY, USA, 1988. ACM Press.

[16] Nancy A. Lynch. Multilevel atomicity a new correctness criterion for database concurrency control. *ACM Trans. Database Syst.*, 8(4):484–502, 1983.

[17] C.H. Papadimitriou. The serializability of concurrent database updates. In *Journal of the ACM 26*, pages 631–653. ACM Press, 1979.

[18] K. Salem, H. Garcia-Molina, and J. Shands. Altruistic locking. *ACM Transactions on Database Systems*, 19, No 1:117–165, 1994.

[19] Clifford A. Shaffer. *A Practical Introduction To Data Structure and Algorithm Analysis*. ALAN APT, Upper Saddle River, NJ 07458, 1997.

[20] Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. Transaction chopping: algorithms and performance studies. *ACM Trans. Database Syst.*, 20(3):325–363, 1995.

[21] Steven S. Skiena. *The algorithm design manual*. Springer-Verlag New York, Inc., New York, NY, USA, 1998.

[22] R.H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. on Database Systems*, 4(2), 180–209.

[23] TPC. Transaction processing performance council.

[24] W. Weihl. Local atomicity properties: Modular concurrency control for abstract data types. *ACM Transactions on Programming Languages and Systems*, 11:249–282, 1989.

[25] Gerhard Weikum and Gottfried Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[26] ANSI X3.135-1992. American national standard for information systems - database language - sql, November 1992.

[27] M. Yannakakis. Serializability by locking. *Journal of the ACM*, 31:227–244, 1984.