A Study on Scheduling Algorithms using Serialization Graph Testing for Distributed Database Systems

Harumasa Tada

March 1998

A Study on Scheduling Algorithms using Serialization Graph Testing for Distributed Database Systems

by

Harumasa Tada

March 1998

Dissertation submitted to Graduate School of the Engineering Science of
Osaka University in partial fulfillment of the requirements for
the degree of Doctor of Philosophy in Engineering

Abstract

This thesis summarizes the works of the author as a master/doctor student of Osaka University from 1993 to 1997 on concurrency control method for distributed database systems.

Concurrency control is one of the key aspect in design of database systems (DBS). Concurrency control is to schedule concurrently executed transactions in order to preserve database consistency. Its objective is to make concurrency of transactions as high as possible. In this thesis, a scheduling algorithm for distributed DBSs is studied so that high concurrency of transactions is obtained without imposing special conditions into transactions.

Our scheduling method is based on Serialization Graph Testing (SGT), which achieves higher concurrency than others. In SGT, a scheduler maintains a directed graph called Serialization Graph (SG) and traverse of the SG is necessary for scheduling. In distributed DBSs, which consist of multiple sites, intersite communication is required for maintenance and traverse of the SG. This communication cost is the most serious problem of SGT in distributed DBSs. In our method, to suppress the communication for maintenance of the SG, the SG is maintained in distributed manner, that is, each site maintains a subgraph of the SG, called a local SG. In this case, an global SG traverse should be done by assembling traverses of local SGs at several sites through message passing. Moreover, we let these local SG traverses be performed simultaneously at several sites to suppress the SG traverse time. Therefore, fractional tags are introduced to detect the completion of global traverse of the SG. Using fractional tags, we proposed the method for the distributed SG traverse.

The correctness of this method is shown and the performance is evaluated by simulations in terms of the number of messages sent for scheduling.

In addition to improvement of the SG traverse, we considered modifying SGT itself to suppress the effect of the scheduling cost to the system performance, We adopted SGT certification because the number of required SG traverses is much smaller in comparison with usual SGT. On the other hand, the delay of abortion in SGT certification may encourage undesirable phenomenons known as cascading aborts. To eliminate cascading aborts, we introduced the idea of another scheduling method, called Optimistic Concurrency Control (OCC), into SGT certification. Thus we proposed a variant of SGT certification so that cascading aborts do not occur. Like OCC, substantial write operations are deferred in the proposed algorithm. The correctness proof and performance evaluation of our algorithm are also presented in this thesis.

In researches of concurrency control in distributed DBSs, SGT has been ignored because of its large scheduling cost. However, considering appearance of new types of databases (e.g. Multimedia databases or Object Oriented databases) in which execution time of transactions tends to become long, high concurrency of SGT should not be ignored. In this thesis, two methods are proposed so that the drawback of SGT is relieved. Simulation results showed that they are useful to suppress the scheduling cost of SGT in distributed DBSs.

List of Publications

(1) Harumasa TADA, Masahiro HIGUCHI, Mamoru FUJII, and Jun OKUI, "A Distributed Scheduling Algorithm Using Serialization Graph Testing with Fractional Tag", *IPSJ Trans.*, vol.38, no.1, pp.90–100, Jan. 1997.

(2) Harumasa TADA, Masahiro HIGUCHI, and Mamoru FUJII, "A Concurrency Control Algorithm Using Serialization Graph Testing with Write Deferring", *IPSJ Trans.*, vol.38, no.10, pp.1995–2003, Oct. 1997.

(3) Harumasa TADA, Masahiro HIGUCHI, and Mamoru FUJII, "A Concurrency Control using Serialization Graph Testing with Write Deferring", in *Proc. of 11th Intl. Conf on Information Networking*, pp. 9C-2.1–9C-2.7, Jan. 1997.

(4) Harumasa TADA, Masahiro HIGUCHI, and Mamoru FUJII, "A Model of Nested Transaction with Fine Granularity of Concurrency Control", in *Proc. of IEEE Pacific Rim Conf. on Communications, Computers and Signal Processing*, pp. 977-980, Aug. 1997.

Acknowledgments

During the course of this work, I have received help from many individuals. First, and foremost, I would like to thank my supervisor Professor Mamoru Fujii for his valuable support and encouragement of the work.

I'm very grateful to Professor Akihiro Hashimoto and Professor Nobuki Tokura for their invaluable comments and helpful suggestions concerning this thesis.

I also wish to thank Assistant Professor Masahiro Higuchi for his valuable suggestions and discussions throughout the work.

I would like to thank Professor Masaru Sudo for his valuable support while I was a bachelor student belonging to Sudo Laboratory of Osaka University.

Many of the courses that I have taken during my graduate career have been helpful in preparing this thesis. I would like to acknowledge the guidance of the late Professor Seishi Nishikawa, and Professor Tadao Kasami, Professor Koji Torii, Professor Kenichi Taniguchi, Professor Hideo Miyahara, Professor Tohru Kikuno, Professor Toshinobu Kashiwabara, Professor Kenichi Hagihara, Professor Katsuro Inoue and Professor Masaharu Imai.

I wish to express my special gratitude to Mr. Yasunori Terasaki for their works on developing the simulator described in Chapter 3.

I also thank to Mr. Masaru Kohara, Mr. Tadashi Yosimoto and Mr. Kazuyuki Uchida for their helpful discussions. I am also grateful to Ms. Yoshimi Katagiri for her kind support.

Finally, I would like to thank the all members of Fujii Laboratory of Department of Informatics and Mathematical Science of Graduate School of Engineering Science of Osaka University.

Contents

1 **Introduction** **1**

2 **Preliminaries** **9**

2.1 Database system overview . . . 9

2.2 Scheduling . . . 11

2.3 Serializability . . . 12

2.3.1 Transactions . . . 13

2.3.2 Histories . . . 14

2.3.3 Conflict Serializability . . . 15

2.3.4 Serialization Graph . . . 16

2.4 Serialization Graph Testing . . . 17

2.4.1 Behavior of Scheduler . . . 17

2.4.2 Addition of Edges to the Serialization Graph . . . 18

2.4.3 Abortion of Transactions . . . 18

2.4.4 Deletion of Unnecessary Nodes . . . 19

2.4.5 Using a Locking Scheme . . . 20

2.4.6 Basic SGT Algorithm . . . 20

2.5 Assumptions . . . 22

3 **A Distributed Graph Traverse Method for Suppressing Communication Cost** **23**

3.1 Introduction . . . 23

3.2 Distributed Serialization Graph Testing . . . 24

3.2.1 Distributed Scheduling . . . 24

3.2.2 Node Storing Site Group . . . 25

3.2.3 Addition of Edges to the Serialization Graph . . . 26

3.2.4 Abortion of Transactions . . . 26

3.2.5 Deletion of Unnecessary Nodes . . . 27

3.2.6 Distributed SGT Algorithm . . . 27

3.3 Fractional Tag Method . . . 28

3.3.1 Traverse of Serialization Graph 28
3.3.2 Fractional Tags 31
3.3.3 Unnecessary Messages 32
3.3.4 Algorithm 33
3.4 Correctness 34
3.4.1 Correctness Proof 34
3.4.2 Livelocks 36
3.5 Evaluation 37
3.5.1 Comparison in Communication Cost 37
3.5.2 Overview of Simulations 41
3.5.3 Simulation Results 42
3.6 Conclusions 42

4 A Scheduling Algorithm for Suppressing Scheduling Cost 45
4.1 Introduction 45
4.2 Certifications 46
4.3 Optimistic Concurrency Control 47
4.4 Proposed Algorithm 48
4.5 Correctness 49
4.6 Evaluation 51
4.6.1 Comparison by Examples 51
4.6.2 Overview of Simulations 54
4.6.3 Simulation Results 55
4.7 Conclusions 59

5 Conclusions 61
5.1 Conclusions 61
5.2 Future Works 62

Chapter 1

Introduction

Concurrency control is one of the key aspect in design on database systems (DBS). Concurrency control is to schedule transactions which are executed concurrently in order to preserve database consistency. Scheduling algorithms are evaluated by the following factors. That is, (1) the concurrency of transaction, (2) the scheduling cost and (3) the ratio of abortions to executed transactions. Of course, one goal of such scheduling algorithms is to provide high concurrency of transactions. The higher the concurrency is, the fewer the cases in which a transaction have to wait for another one are. Transactions which do not have to wait for other transactions are processed quickly and therefore the throughput of DBSs increases. Reducing the scheduling cost is also significant because the time for scheduling directly affects the execution time of each transaction. In *distributed database systems* (distributed DBSs) which consist of several *sites*, intersite communication is required for scheduling. In the case of distributed DBSs, this communication cost is also taken into account as the scheduling cost. Some transactions may be aborted for scheduling. The abortion of a transaction means waste of the time for which the transaction executed. If the wasted execution time increases, then the throughput of DBSs decreases. Hence it is desirable that the ratio of abortions is small.

Serializability is commonly used as a criterion of validity for concurrent execution of transactions. Many researchers proposed lots of scheduling algorithms based on serializability. The proposed scheduling algorithms can be classified into two types, that is, *locking protocols* which locks data items for scheduling, and *non-locking protocols* which do not lock data items.

Two Phase Locking (2PL) is the simplest and most well-known locking protocol. In 2PL, before a transaction T accesses a data item x, T must obtain the lock of x. If another transaction T' is holding the lock, T has to wait until

T' releases the lock. Moreover, 2PL imposes a rule which is called *two phase rule*, that is, once a transaction T has released a lock, T should not subsequently obtain any more locks. The main cost for 2PL is maintenance of a table which is called the *lock table.* Thus 2PL is easy to implement and its scheduling cost is small. This is the merit of 2PL. The shortcoming is that transactions are *blocked* in 2PL. That is, some transactions stop waiting for other transactions. Due to the transaction blocking, the concurrency provided by 2PL is not so high. Some blocked transactions often form chains and the chains may become long. Such a long chain is undesirable because the transaction at the end of the chain should wait for many transactions. Moreover, some blocked transactions may cause *deadlocks.* Hence some deadlock detection scheme is needed for 2PL. When a deadlock is detected, one transaction involved in the deadlock should be aborted to solve a deadlock. However, deadlock is the only case in which 2PL aborts transactions. Therefore, the number of abortions in 2PL is relatively small. Since 2PL is the most common scheduling algorithm, a lot of 2PL variants have been proposed to overcome the drawback of 2PL—low concurrency.

Some 2PL variants use a priori knowledge of transactions [8][9]. They require each transaction to predeclare the readset and the writeset, i.e., the set of data items which the transaction intends to read and write. Since a scheduler knows which data items each transaction intends to access, locks held by the transaction can be released sooner. Nevertheless, it is very difficult for many transactions to predict the readset and the writeset precisely before execution. Hence the cases in which such algorithms are available are limited.

There are many 2PL variants which use semantic knowledges of transactions [1][2][11][13][21][31]. In algorithms proposed in [2], [11], [13] and [21], each transaction has several breakpoints at which other transactions can interleave. By using semantic knowledges, some non-serializable executions can be permitted by such algorithms. However, users who design transactions must specify what type of transactions can interleave the transaction which he wants to execute. This imposes a heavy load on users. Moreover, these algorithms have the potential risk of users' mistakes in their specifications. In other words, the reliability of DBSs may be spoiled under these algorithms. In [1] and [31], abstract atomic operations (e.g. deposit, withdraw and member operations) is introduced in addition to primitive read and write operations. Since commutativity of such abstract operations are higher than that of primitive ones, higher concurrency is obtained. However, atomicity of such operations depends on implementation of database systems. Furthermore, there are different types of 2PL variants [12][27].

Locking protocols are studied much more actively than non-locking protocols. Practically, they are widely used in many applications. Nevertheless, the problems of transaction blocking and deadlocks are inevitable in locking protocols. To eliminate it perfectly, we should introduce non-locking protocols.

Non-locking protocols do not lock any data items and therefore no transactions are blocked. Instead, some transactions may violate database consistency. Such transactions are aborted and all their effects are wiped out. Aborted transactions are restarted later. Therefore, the ratio of abortions may become larger than that in locking protocols.

In compensation for release from the blocking problems, non-locking protocols suffer from the phenomenon which is called *cascading abort.* In locking protocols, a transaction T cannot read a data item x which is locked by another transaction. In most implementations of locking protocols, transactions release their locks at the time of commitment. Hence there exist no cases in which a transaction reads a data item which was updated by another uncommitted transaction. On the other hand, under non-locking protocols, a transaction T can read a data item x updated by an uncommitted transaction. Although this feature increases the concurrency, it also causes the problem. Suppose that T read a data item x which was updated by T'. When T' is aborted, the value of x which T read becomes invalid. Hence T should be aborted too. In this way, an abortion of a transaction may cause other abortions. This phenomenon is called cascading abort. Cascading abort is a problem because it can lead to a large number of abortions.

While most locking protocols are variants of 2PL, there exist several types of non-locking protocols. *Timestamp Ordering* (TO) is the most common non-locking protocol. Instead of locking data items, TO uses timestamps for scheduling. In TO, each transaction is assigned a unique timestamp. Transactions are scheduled according to the total order of timestamps. Before the execution of an operation o of a transaction T, the scheduler compares the timestamp of T with the timestamp of T' whose previously executed operation o' conflicts with o. If the timestamp of T is larger than that of T' (i.e. T is newer than T'), then o is executed. Otherwise o is rejected and T is aborted. The scheduling cost of TO is not so large. It consists of generation and comparison of timestamps. TO's drawback about concurrency is that the total order of timestamps, which are assigned *before* executing transactions, restricts concurrent execution of transactions. If the TO scheduler could foresee future execution of transactions, it could assign the timestamp according to the order that the execution will be serialized. In this case, all serializable executions would be permitted. Practically, the TO scheduler cannot know about the future, and it may assign timestamps which conflict with serialization order of future execution. Thus some serializable executions are prohibited in TO. Several TO variants has been proposed [20][26]. TO and its variants avoid the blocking problem of 2PL, which is the greatest advantage of TO. However, it does not necessarily means that TO provides higher concurrency than 2PL. The class of concurrent executions permitted under TO does not include that under 2PL and vice versa. Therefore, it is difficult to compare the concurrency provided by TO with that provided by 2PL.

Though algorithms based on 2PL or TO are simple and their scheduling cost is relatively small, they excluded some serializable executions for simplicity of scheduling. Hence they cannot permit all serializable executions. This is a limit of concurrency provided by variants of 2PL and TO. As mentioned above, to achieve high concurrency with these approaches, some special condition is needed on DBSs.

So far, another non-locking protocol called *Serialization Graph Testing* (SGT) [4] has not received much attention. SGT uses a directed graph which is called *serialization graph* (SG) for scheduling. In SGT, database consistency is preserved by ensuring that the SG remains acyclic. An SGT scheduler maintains the SG which represents the relative order of the executed operations. The scheduler traverses the SG before the execution of an operation o, and if o does not cause any cycle to the SG, o is executed immediately, otherwise o is rejected. The feature of SGT is that all serializable executions are permitted under SGT. Therefore, SGT provides higher concurrency of transactions than 2PL and TO [4]. Since SGT uses abortions for scheduling as TO does, the ratio of abortions may be larger than that of 2PL. Due to high concurrency, however, executions which are rejected in TO may be permitted in SGT. Hence the ratio of abortions is smaller than TO. The most serious defect of SGT is its scheduling cost. SGT requires traverse of a directed graph for each operation and it needs much cost. The recent studies of SGT are scarcely reported. Badrinath et al. [3] discussed about abstract operations to increase concurrency of SGT. *Cautious Scheduler* [16] uses the *transaction IO graph* which is a variant of the SG.

At last, *Certifications* ,which are most aggressive in non-locking protocols, should be mentioned. In usual scheduling methods, before executing each operation, it is checked whether the operation preserves the database consistency or not. In Certifications, however, all operations of transactions are executed immediately. At the time of commitment of a transaction, the consistency check for the transaction is done. If it has succeeded, the transaction is committed. Otherwise the transaction is aborted.

There are some Certifications using different ways of the consistency check. Locking based Certification is discussed in [14]. Despite its name, it does not really lock data items. It uses *optimistic locks* for scheduling. When a transaction T accesses the data item x, which takes the optimistic lock of x. Note that T can take it even if it is held by another transaction. In validation of T, if optimistic locks held by T conflicts any optimistic lock held by another executing transaction T', T is aborted. Otherwise T is committed. Timestamp based Certification is studied by many researchers [5][7][14][28][32]. SGT based Certification is discussed in [4][6][32]. We call it *SGT certification.* Certification approach seems to be suitable for SGT, because consistency is checked only when transactions are to be committed while usual SGT checks consistency for each operation. *Optimistic Concurrency Control* (OCC) [19] uses unique method for validation. In OCC, a transaction consists of three phases, the read phase, the validation phase and the write phase. Basically, the validation of OCC is the same as that of Timestamp based Certification. However, OCC is more restrictive and therefore concurrency provided by OCC is lower than Timestamp based Certification. The advantage of OCC is that it can avoid cascading aborts.

The feature of Certifications is that validation is done only once for each transaction. In scheduling algorithms other than Certifications, it should be made sure that each operation preserves database consistency before it is executed. That is, validation is done for each operation. Hence the scheduling cost in Certifications is much smaller than others. The drawback is that conflicts are not detected until a transaction is about to commit and therefore the abortion is delayed. If the abortion of a transaction is delayed, then the waste of execution time becomes larger. Moreover, this delay of abortions increases the possibility of cascading abort.

As described above, lots of scheduling algorithms have been proposed so far. Nevertheless, due to the recent development of databases and their applications, new scheduling algorithm is required.

In traditional database applications, for example, banking systems, seat reservation systems in airlines and so on, the task for each transaction is simple. Therefore, the execution time of transactions are generally short. In such DBSs, the ratio of the time for scheduling in the execution time of transaction is large. Thus reducing the scheduling cost is more important than increasing the concurrency of transactions and avoiding abortions. However, circumstances around DBSs are changing nowadays. First, the performance of computers has improved remarkably. Hence the relative costs of computations in DBSs are decreasing rapidly. Second, new types of databases are appearing in recent years, for example, Multimedia Databases, Object Oriented Databases and so on. One of the features of such databases is that the execution time of transactions tends to become long. Such transactions are called *Long-lived transactions* (LLT).

As circumstances change, the demand on concurrency control is also changing. That is, high concurrency and few abortions are becoming more important than low scheduling cost. The scheduling cost is almost computational, for example, updating the lock table, comparing the value of two timestamps, traverse the SG and so on. By improvement of performance of the MPUs, memories and so on, the effect of the scheduling cost to the performance of DBSs has become small, and we can expect that it will become smaller in future. In DBSs in which LLTs exist, the relative effect is much smaller. On the other hand, the concurrency affects the execution time of transactions in a different way. It is related to the time for which transactions wait for other transactions. The number of abortions affect the wasted execution time. The execution time of transactions includes the access time of mechanical devices (e.g. hard disk drives), the communication delay for data transmission, the waiting time for user interaction and so on. Clearly, such kinds of times are very difficult to reduce drastically.

In this thesis, we attach more importance to the concurrency of transactions and the number of abortions than the scheduling cost. In spite of large scheduling cost, we chose SGT as the scheduling algorithm because SGT permits all serializable executions. Furthermore, in order to develop the scheduling algorithm which is useful in general cases, it seems to be worthiness to eliminate any kind of a priori knowledges, semantic knowledges and abstract operations.

When SGT is applied to distributed DBSs, the communication cost is the most serious difficulty. In distributed DBSs, SGT needs much intersite communications for checking acyclicity of the SG because of the global structure of the SG. This communication cost accounts for a large part of the scheduling cost of SGT in distributed DBSs.

In this thesis, two approaches to overcome above difficulty of SGT are proposed and evaluated.

In Chapter 2, the preliminaries of the thesis is described. Overview of our model of database system, some formal definitions and the basic SGT algorithm are included.

To suppress the communication cost of distributed SGT, the method for traverse of the SG in distributed DBSs is proposed in Chapter 3. We call the method *Fractional Tag Method* (FT). The aim of FT is suppression of the cost of SG traverse in distributed DBSs. Transactions in distributed DBSs are classified into two types, that is, *local* transactions, which access its local database only, and *global* transactions, which access several remote databases. As Özsu et al. have pointed out in [24], most distributed DBSs are structured to gain maximum benefit from data localization. Hence local transactions are dominant in most distributed DBSs. The problem is that the distributed SGT scheduling needs some communications even for local transactions. FT suppresses this communication cost for local transactions. Therefore, FT is useful in distributed DBSs in which most transactions are local ones. In Chapter 3, we show the correctness of FT and evaluate its performance by simulations in terms of the number of messages which are needed for scheduling.

To suppress the effect of the scheduling cost to the system performance, a variant of SGT is proposed in Chapter 4. Even if FT is used, the communication cost for scheduling is still large especially in global transactions. To alleviate the effect of the scheduling cost, we adopted SGT certification. Under SGT certification, a scheduler traverses the SG only once for each transaction. Thus the scheduling cost of SGT is suppressed. On the other hand, the delay of abortion increases the possibility of cascading aborts. To solve the problem of cascading aborts, we exploited the fact that OCC [19] avoids cascading aborts. The interesting feature of OCC is that it first performs write operations to internal buffers, and the values are then written to the actual database at the termination of the transaction that wrote them. Since the values in the buffers cannot be accessed by other transactions, write operations are deferred practically. We apply this feature of OCC to SGT certification and propose a scheduling algorithm. In our algorithm, substantial write operations are deferred as in OCC. Therefore, we call it *SGT with Write Deferring* (SGT-WD). The feature of SGT-WD is (1) no cascading aborts occur, (2) the number of abortions is suppressed, and (3) data restoring is unnecessary when transaction is aborted for scheduling. In Chapter 4, the SGT-WD algorithm and its correctness are presented. Moreover, the performances of SGT-WD, SGT, and SGT certification are evaluated by simulations on distributed database systems.

In Chapter 5, a summary of the methods and their results described in this thesis are presented, and future works are discussed.

Chapter 2

Preliminaries

2.1. Database system overview

The main component of a database system model is a *transaction*. Informally, a transaction is an execution of a program that accesses a shared database. The goal of concurrency control is to ensure that transactions execute *atomically*, meaning that

1. each transaction accesses shared data without interfering with other transactions, and

2. if a transaction terminates normally, then all of its effects are made permanent; otherwise it has no effect at all.

A *database* consists of a set of named *data items*. Each data item has a value. The values of the data items represent a *state* of the database. The size of the data contained in a data item is called the *granularity* of the data item. Granularity will usually not be significant to our study and therefore it will be left unspecified.

A *database system* (DBS) is a collection of hardware and software modules that support commands to access the database, called *database operations*, or simply operations. The most important operations are *Read* and *Write*. It is assumed that a *Read* (or *Write*) operation can access only one data item. To access multiple data items, multiple operations are required.

The DBS also supports *transaction operations*: *Start*, *Commit*, and *Abort*. A program tells the DBS that it is about to begin executing a new transaction by issuing the operation *Start*. It indicates the termination of the transaction by issuing either operation *Commit* or *Abort*. By issuing a *Commit*, the program tells the DBS that the transaction has terminated normally and all of its effects should be made permanent. By issuing an *Abort*, the program tells the DBS

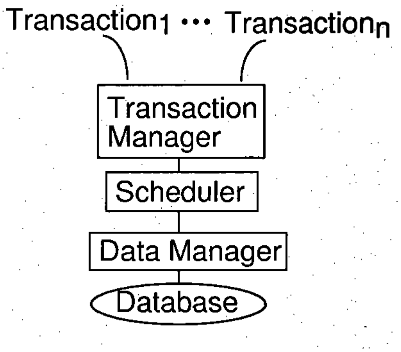

Figure 2.1: A system configuration of a database system

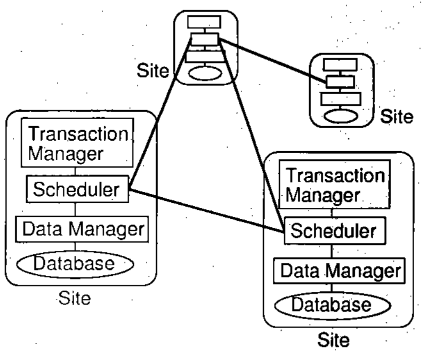

Figure 2.2: A distributed database system.

that the transaction has terminated abnormally and all of its effects should be obliterated.

A DBS consists of three *modules* : a *transaction manager*, which performs any required preprocessing of database operations and transaction operations it receives from transactions ; a *scheduler*, which controls the relative order in which database operations and transaction operations are executed; and a *data manager* which operates directly on the database.

Database operations and transaction operations issued by a transaction to the DBS are first received by the transaction manager. The operations then move down through the scheduler, and data manager. Thus, each module sends requests to and receives replies from the next lower level module. Figure 2.1 shows a system configuration of a database system.

A *distributed database system* (distributed DBS) is composed of multiple DBSs. A distributed DBS is a collection of *sites* connected by a communication network. Each site is a DBS, which stores a portion of the database. Figure 2.2 shows a system configuration of a distributed DBS.

In distributed DBSs, transactions are classified into two types.

- Local transactions: which access only data items stored in the site that executes it.

- Global transactions: which access data items stored in more than one sites.

To consider the scheduling algorithm that preserves database consistency, consistent states of the database must be defined. It is assumed that

1. the initial state of a database is consistent, and

2. any transaction preserves consistency of a database if it is executed alone.

Thus, all states obtained by executing transactions serially from the initial state are consistent and others are not.

Since recovery problems are not discussed in this thesis, it is assumed that

1. there are no cancellations of transactions by a user,

2. there are no errors in any transaction programs, and

3. any type of system failure does not occur.

2.2. Scheduling

A scheduler is a program or collection of programs that controls the concurrent execution of transactions. It performs this control by restricting the order in which the data manager executes Reads, Writes, Commits, and Aborts of different transactions. Its goal is to order these operations so that the resulting execution preserves consistency of the database.

To execute a database operation, the transaction manager passes the operation to the scheduler. After receiving the operation, the scheduler can take one of three actions:

1. Execute: The scheduler can pass the operation to the data manager. When the data manager finishes executing the operation, it sends a reply to the scheduler. Moreover, if the operation is a Read, the data manager returns the value(s) it read, which the scheduler relays back to the transaction.

2. Reject: The scheduler can refuse to process the operation, in which case it tells the transaction manager that its operation has been rejected. This causes the transaction to abort.

3. Delay: The scheduler can delay the operation by placing it in a queue internal to the scheduler. Later, it can remove the operation from the queue and either execute it or reject it. In the interim (while operation is being delayed), the scheduler is free to schedule other operations.

Using its three actions – executing an operation, rejecting it, or delaying it – the scheduler control the order in which operations are executed. When it receives an operation from the transaction, it checks whether the operation can be executed without violating the database consistency or not. We call this consistency check the *validation.* If the validation succeeded, then it passes the operation to the data manager right away. Otherwise, it either delays the operation (if it may be able to correctly process the operation in the future) or reject the operation (if it will never be able to correctly process the operation in the future). Thus it uses execution, delay and rejection of operations to help produce correct executions.

The scheduler is quite limited in the information it can use to decide when to execute each operation. It is assumed that it can only use the information that it obtains from the operations that transactions submit. The scheduler does *not* know any details about the programs comprising the transactions, except as conveyed to it by operations. It can predict neither the operations that will be submitted in the future nor the relative order in which these operations will be submitted. When this type of advance knowledge about programs or operations is needed to make good scheduling decisions, the transactions must explicitly supply this information to the scheduler via additional operations. Unless stated otherwise, we assume that such information is not available.

2.3. Serializability

When two or more transactions execute concurrently, their database operation execute in an interleaved fashion, i.e. operation from one transaction may execute between two operations from another transaction. This interleaving can cause transactions to behave incorrectly, or *interfere*, thereby leading to an inconsistent database. This interference is entirely due to the interleaving. That is, it can occur even if each transaction program is coded correctly and no component of the system fails. The goal of concurrency control is to avoid such interference and thereby preserve database consistency.

2.3.1 Transactions

Transaction is a particular execution of a program that manipulates the database by means of *Read* and *Write* operations. Formally, a transaction is a representation of an execution of the *Read* and *Write* operations and indicates the order in which these operations are to be executed. For each *Read* and *Write*, the transaction specifies the names, but not the values, of the data items read and written respectively. In addition, the transaction contains a *Commit* or *Abort* as its last operation, to indicate whether the execution was terminated successfully or not.

In general, a notation $r_i[x]$ (or $w_i[x]$), where x is a data item, is used to denote *Read* (or *Write*) operation issued by the transaction T_i on a data item x. To keep this notation unambiguous, it is assumed that no transaction reads (or writes) a data item more than once. Similarly, c_i and a_i is used to denote *Commit* and *Abort* operations of T_i (respectively). In a particular transaction, only one of these two can appear. The arrows indicate the order in which operations execute. Thus in the example $r_i[x] \rightarrow w_i[x] \rightarrow c_i$, $w_i[x]$ follows ("happens after") $r_i[x]$ and precedes ("happens before") c_i.

A transaction may represent concurrent execution of programs. Therefore, a transaction is modeled as a partial order.

The definition of a transaction is formalized as a partial ordering of operations. The name of the partial order (i.e. transaction name T_i) is used to denote not only the partial order itself but also the set of operations in the partial order. The meaning of a symbol that denotes both a partial order and its elements will always be clear from the context. In particular, when $r_i[x] \in T_i$, means that $r_i[x]$ is an element (i.e. operation) of T_i, T_i denotes the set of operations in the partial order. The formal definition is following.

Definition 2.1 Let DB be a set of all data items in a database system. A *transaction* T_i is a partial order with the ordering relation $<_i$ where
(1) $T_i \subseteq \{r_i[x], w_i[x] \mid x \subseteq DB\} \cup \{a_i, c_i\}$;
(2) $a_i \in T_i$ iff $c_i \notin T_i$;
(3) if t is c_i or a_i (whichever is in T_i), for any other operation $p \in T_i$, $p <_i t$; and
(4) if $r_i[x], w_i[y] \in T_i$ and $x \cap y \neq \phi$, then either $r_i[x] <_i w_i[y]$ or $w_i[y] <_i r_i[x]$. □

Figure 2.3 shows examples of transactions.

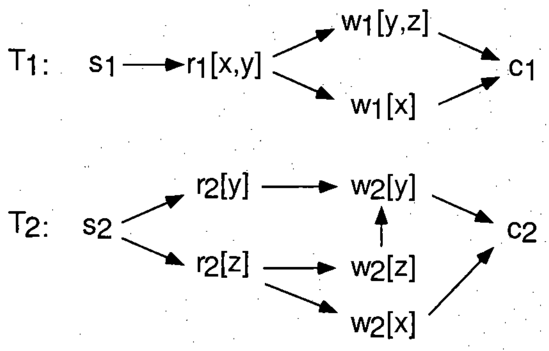

Figure 2.3: Examples of transactions

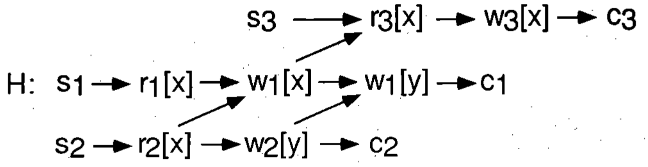

Figure 2.4: An example of a history

2.3.2 Histories

A history indicates an order in which operations of transactions were executed relative to each other. Since some of these operations may be executed in parallel, a history is defined as a partial order.

Definition 2.2 Two operations are said to be *conflict* if they both operate on a same data item and at least one of them is a *Write*. □

Definition 2.3 For two partial orders H and H' (with ordering relation $<_H$ and $<'_H$ respectively), H' is said to be a *prefix* of H if $p <_H q$ for any $p, q \in H'$ such that $p <_{H'} q$. □

Definition 2.4 Let $T = \{T_1, T_2, ..., T_n\}$ be a set of transactions. A *complete history* H over T is a partial order with ordering relation $<_H$ where:
(1) $H = \cup_{i=1}^{n} T_i$;
(2) $<_H \supseteq \cup_{i=1}^{n} <_i$; and
(3) for any two conflicting operations $p, q \in H$, either $p <_H q$ or $q <_H p$.
Any prefix of a complete history is called a *history*. □

Figure 2.4 shows an example of a history.

Thus a history represents a possibly incomplete execution of transactions. A failure may interrupt the execution of active transactions. Therefore, arbitrary histories must be considered, not merely complete ones.

A transaction T_i is said to be *committed* (or *aborted*) in history H if $c_i \in H$(or $a_i \in H$). T_i is said to be *active* in H if it is neither committed nor aborted.

Definition 2.5 For a history H, the *set of possible complete histories* of H, denoted $PH(H)$, is the set of all complete histories which include H as prefix. □

A dot($\cdot$) represents connection of a operation to a history. Let H be a history and o be a operation. If history $H' = H \cdot o$, then for any operation $o_i \in H$, $o_i <_{H'} o$.

2.3.3 Conflict Serializability

The goal of scheduling is to produce histories which preserve database consistency. The simplest way to preserve database consistency is to execute transactions serially. However, this would mean that the database system could not execute transactions concurrently. Without such concurrency, the system may make poor use of its resources, and so might be too inefficient. Therefore, the class of allowable executions is broadened to include executions that have the same effect as serial ones. Such executions are called *view serializable* [4]. The set consists of all view serializable histories is called *class* VSR. View serializable histories preserve database consistency. Therefore, all view serializable histories may be accepted by the scheduler. Nevertheless, no scheduling algorithms that accept all view serializable histories are known, because it is difficult to examine whether a given history is view serializable or not. In fact, it is proven that testing a history for view serializability is NP-complete [25].

Therefore, the subclass of VSR called *conflict serializability* (CSR) [4] is introduced as a set of histories which may be allowed by schedulers. Before giving the definition of conflict serializability, *conflict equivalence* is defined.

Definition 2.6 Two histories H_1 and H_2 are *conflict equivalent* iff
(1) H_1 and H_2 are defined over the same set of transactions and have the same operations; and
(2) H_1 and H_2 order conflicting operations of non-aborted transactions in the same way; that is, for any conflicting operations p_i and q_j belonging to transactions T_i and T_j (respectively) where $a_i, a_j \notin H_1$, if $p_i <_{H1} q_j$ then $p_i <_{H2} q_j$ (this implies: $p_i <_{H1} q_j$ iff $p_i <_{H2} q_j$) □

Definition 2.7 A history H is *conflict serializable*(CSR) iff one of the element of $PH(H)$ is conflict equivalent to a serial history H_s. □

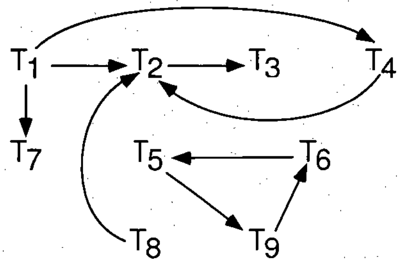

Figure 2.5: An example of a serialization graph

In the literature, the word serializable (SR) usually means CSR. In the rest of this thesis, the word serializable (SR) is used instead of conflict serializable.

2.3.4 Serialization Graph

The (conflict) serializability of a history is determined by analyzing a graph derived from the history called a *serialization graph* (SG). A serialization graph is a directed graph derived from a history, and has one node for each transaction. For simplicity, a transaction name is also regarded as the name of the node for the transaction.

Definition 2.8 For a history H, the *serialization graph* $SG(H) = \langle V, E \rangle$ is a directed graph such that

$V = \{T_i \mid T_i$ is a transaction that is already commuted in $H\}$

$E = \{(T_i, T_j) \mid$ there exists conflicting operations $o_i \in T_i$ and $o_i \in T_j$ such that $o_i <_H o_j$ where $T_i, T_j \in V\}$ □

Figure 2.5 shows an example of a serialization graph.

Each edge (T_i, T_j) in $SG(H)$ means that at least one of T_i's operations precedes and conflicts with one of T_j's. This suggests that T_i should precede T_j in any serial history that is equivalent to H. If a serial history H_s which is consistent with all edges in $SG(H)$ is found, then H_s is conflict equivalent to H and therefore H is SR. This is possible as long as $SG(H)$ is acyclic. The following theorem was shown in [4].

Theorem 2.1 A history H is SR iff $SG(H)$ is acyclic. □

The reachability of nodes in a directed graph is defined as follows.

Definition 2.9 A node T is *reachable* from a node T_i in a directed graph G iff there is a directed path from T_i to T in G. □

Note that T is not reachable from T itself unless a directed path from T to T exists in G.

2.4. Serialization Graph Testing

Serialization Graph Testing (SGT) is a method for controlling the order in which operations are executed in order to preserve database consistency.

In SGT, validation is done using the serialization graph (SG). A scheduler maintains the SG of a history that represents the execution it controls. As the scheduler sends new operations to the data manager, the history changes and the SG maintained by the scheduler also changes. A scheduler attains SR executions by ensuring that the SG it maintains always remains acyclic.

Suppose a scheduler has an acyclic SG. When it receives an operation o, it begins the validation for o and checks the acyclicity of the SG assuming that o has been executed. If the SG is acyclic, then the validation is successful. Otherwise it is failed. According to the result of the validation, the scheduler executes, rejects, or delays the operation o.

2.4.1 Behavior of Scheduler

From the definition in section 2.3.4, an SG contains nodes for all committed transactions and no others. Such an SG differs from that usually maintained by a scheduler, because the latter usually contains nodes for all active transactions, which are not yet committed. Therefore, *serialization graph with active transactions* (SGA), which is defined as follows, is introduced.

Definition 2.10 For a history H, the *serialization graph with active transactions* $SGA(H) = \langle V, E \rangle$ is a directed graph such that

$V = \{T_i \mid T_i$ is active or committed transaction in $H\}$

$E = \{(T_i, T_j) \mid$ two operations $o_i \in T_i$ and $o_i \in T_j$ conflict such that $o_i <_H o_j$ where $T_i, T_j \in V\}$ □

A scheduler maintains an SGA instead of an SG. Suppose a scheduler receives an operation $p_i[X]$ from the transaction manager. If a node for T_i does not yet exist in its SGA, then the scheduler first adds the node in its SGA. Next, it adds an edge from T_j to T_i for every previously scheduled operation $q_j[Y]$ that conflicts with $p_i[X]$. Then the scheduler traverses the SGA and checks acyclicity of it. Two cases are possible:

1. The resulting SGA contains a cycle. This means that if $p_i[X]$ were to be scheduled now (or at any point in the future), the resulting execution would be non-SR. In this case, the scheduler rejects $p_i[X]$ and T_i is aborted. It sends a_i to the data manager and, when a_i is acknowledged, it deletes T_i and all edges incident with T_i from the SGA. Deleting T_i makes the SGA

acyclic again, since all cycles that existed involved T_i. Since the SGA is acyclic, the execution produced by the scheduler – with T_i aborted – is SR.

2. The resulting SGA is still acyclic. In this case, the scheduler can accept $p_i[X]$. It can schedule $p_i[X]$ immediately, if all conflicting operations previously scheduled have been acknowledged by the data manager; otherwise, it must delay $p_i[X]$ until the data manager acknowledges all conflicting operations.

2.4.2 Addition of Edges to the Serialization Graph

The *readset* and *writeset* of transaction T are defined as follows.

Definition 2.11 For a transaction T and a history H, $readset(T, H) = \{x \mid T$ has read from x in $H\}$ $writeset(T, H) = \{x \mid T$ has written to x in $H\}$

That is, they are the sets of data items that have been read and written by T in history H. In SGT, a scheduler detects conflicts between transactions in order to add edges to the SGA. To detect conflicts, for each transaction T, the scheduler must maintain the readset and writeset of T. In a scheduling algorithm, $readset(T)$ and $writeset(T)$ stores $readset(T, H)$ and $writeset(T, H)$ respectively where H is the history produced so far. When an operation o of a transaction T reads (writes) a data item x, x is added to $readset(T)$ ($writeset(T)$). For a transaction T_i such that $x \in writeset(T_i)$ ($x \in readset(T_i) \cup writeset(T_i)$), an edge from T_i to T is added to the SGA.

2.4.3 Abortion of Transactions

Described in section 2.4.1, a transaction may be aborted by failure of validation. When a transaction aborts, the DBS must wipe out its effects. The effects of a transaction T are of two kinds: (1) effects on data, that is, values that T wrote in the database; and (2) effects on other transactions, namely, transactions that read values written by T. Both should be obliterated.

The DBS should remove T's effects by restoring, for each data item x updated by T, the value x would have had if T had never taken place. I say that the DBS *undoes* T's Write operations. Maintaining the restoring value involves a complex problem.

The DBS should remove T's effects by aborting the affected transactions. Aborting these transactions may trigger further abortions, a phenomenon called *cascading abort*.

2.4.4 Deletion of Unnecessary Nodes

In section 2.4.1, the SGA contains nodes for all active transactions and all committed transactions. Of course, the number of committed transaction increases as time goes on. That is, the size of the SGA grows very large and so does the cost of maintenance and traverse of the SGA. Therefore, the nodes which are no longer necessary for scheduling, should be deleted from the SGA. An unnecessary node for scheduling is a node which is never involved in any cycle that will be produced in the future. Suppose that a history H contains transaction T_i committed by operation c_i. Since all operations in T_i have already finished, all transactions containing a operation executed after c_i must appear after T_i in any serial histories which is conflict-preserving equivalent to H. Therefore, no edges incoming to node T_i are added to the SGA by operations executed after c_i. When T_i is committed, if there are no edges incoming to T_i, no cycles involving T_i will be produced in the future. In this case, node T_i and all edges outgoing from T_i can be deleted from the SGA. Deleting these node and edges may trigger another deletions. As a result of deletion of edges outgoing from T_i, if all edges incoming to another node T_j are lost, then T_j and all edges outgoing from T_j can be deleted. I call this phenomenon *cascading node deletion.* In this way, the node for committed transaction is sure to be deleted at some time, and the size of the SGA does not grow so large. A subgraph of a SGA made by deletion of all unnecessary nodes is called a *stored serialization graph* (SSG).

Definition 2.12 For a history H, the *stored serialization graph* $SSG(H) = \langle V, E\rangle$ is the maximum subgraph of $SGA(H)$ such that

$V = \{T_i \mid T_i$ is committed in H and T_i has incoming edges or T_i is active in $H\}$

$E = \{(T_i, T_j) \mid$ two operations $o_i \in T_i$ and $o_i \in T_j$ conflict such that $o_i <_H o_j$ where $T_i, T_j \in V\}$ □

Theorem 2.2 A history H is SR iff $SSG(H)$ is acyclic.

[Proof] (if) Suppose that $SSG(H)$ is acyclic. $SGA(H)$ is also acyclic because any node $T_i \in SGA(H)$ where $T_i \notin SSG(H)$ has no incoming edges. Since $SG(H) \subseteq SGA(H)$, $SG(H)$ is also acyclic. By theorem 1, H is SR.

(only if) Suppose that a history H is SR. There exists a complete history $H_c \in PH(H)$ such that H_c is conflict-preserving equivalent to a serial history. Clearly H_c is SR. From theorem 1, $SG(H_c)$ is acyclic. Since H_c is a complete history, $SG(H_c) = SGA(H_c)$. That is, $SGA(H_c)$ is acyclic too. Because H contains H_c as prefix, $SGA(H) \subseteq SGA(H_c)$. Therefore, $SGA(H)$ is acyclic. Since $SSG(H) \subseteq SGA(H)$, $SSG(H)$ is acyclic too. □

In usual SGT algorithm, an SSG is used for scheduling instead of an SG. For simplicity, the stored serialization graph (SSG) is referred as the serialization graph (SG) in the rest of this thesis.

2.4.5 Using a Locking Scheme

Under SGT, when a transaction T reads or writes a data item x, the following processes are executed:.

- Edges are added to the SG.
- x is added to *readset*(T) (*writeset*(T)).
- x is read from (written to) the actual database.

If the above processes are interleaved with other conflicting operations, the SG maintained by the scheduler may contradict with $SG(H)$. For example, suppose that a transaction T_1 tries to write a data item x. *writeset*(T_1) has been updated, but the value of x has not been written into the actual database when another transaction T_2 reads x. The edge from T_1 to T_2 is then added to the SG according to *writeset*(T_1). However, T_2 reads the value of the "old" x that has not been updated by T_1. The edge from T_1 to T_2 contradicts with the fact that T_2 reads x before T_1 writes it. In order to avoid such a case, we use a locking scheme. Before accessing a data item x, the readlock (writelock) of x is held by o to prohibit the execution of operations that conflict with o (line 2 of read phase and line 2 of validation phase). Then, the above processes are executed. The lock is released when o finishes reading (writing) x. The locking scheme used here is different from that used in two-phase locking (2PL). Under 2PL, the lock of a data item x is held by a transaction T when T accesses x, and is not released until T holds all required locks. Thus, the locking time depends on the internal processing of transactions. In the case of long-lived transactions, the locking time may be very long. On the other hand, under SGT, the lock is held during the SG updating and the data accessing. The locking time is independent of the internal processing of transactions. Therefore, the influence of our locking scheme on the concurrency of transactions is much smaller than that of 2PL.

2.4.6 Basic SGT Algorithm

Table 2.1 shows the SGT algorithm. The SG maintained by a scheduler is denoted as SG in Table 2.1.

Table 2.1: The SGT algorithm

Basic algorithm of SGT

```
if an operation $o$ of transaction $T$ is received from the transaction manager (TM) then
    if $o = start$ then
        add a node $T$ to $SG$
        start $T$
        send a reply such that $T$ is started to the TM
    else if $o = commit$ then
        commit $T$
        delete unnecessary nodes from $SG$
        send a reply such that $T$ is committed to the TM
    else /* $o = read(x)$ or $o = write(x)$ */
        if $o = read(x)$ then
            set the readlock of $x$ to $T$
            for each $T_j$ such that $x_i \in writeset(T_j)$ do
                add an edge $T_j \rightarrow T$ to $SG$
            add $x$ to $readset(T)$
        else /* $o = write(x)$ */
            set the writelock of $x$ to $T$
            for each $T_j$ such that $x_i \in readset(T_j) \cup writeset(T_j)$ do
                add an edge $T_j \rightarrow T$ to $SG$
            add $x$ to $writeset(T)$
        if $SGT\text{-}validation(T) = true$ then
            if $o = read(x)$ then
                read $x$ from the database
                release the readlock of $x$
            else /* $o = write(x)$ */
                write $x$ to the database
                release the writelock of $x$
            send a reply such that $o$ succeeded to the TM
        else /* $SGT\text{-}validation(T) = false$ */
            $Abort \leftarrow \{T\}$
            while there exists a transaction $T_i \notin Abort$ which read from $T_j \in Abort$ do
                $Abort \leftarrow Abort \cup \{T_i\}$
            for each $T_j \in Abort$ do
                abort $T_j$
                delete $T_j$ from $SG$
                delete unnecessary nodes from $SG$
            send a reply such that all transactions in $Abort$ are aborted to the TM
```

Note: *SGT-validation*(T) is a function such that it returns *true* if a serialization graph SG has no cycles including T and otherwise it returns *false*.

2.5. Assumptions

In this thesis, we discuss the scheduling method for distributed DBSs. Assumptions of distributed DBSs we deal with are described as follows.

- Each site is connected by a large scale network. That is, the communication delay cannot be ignored and message broadcasts take large costs.
- Each site knows locations of all of data items. When a transaction T in a site s accesses a data item stored in another site s', T can directly access s' because s knows that the data item is stored in s'
- The number of sites included in a distributed DBS is not so large (about 10 sites).
- Most of transactions are local transactions.
- Most of global transactions do not access so many sites.

Chapter 3

A Distributed Graph Traverse Method for Suppressing Communication Cost

3.1. Introduction

In this chapter, the method for traverse of the SG in distributed DBSs is proposed. It suppress the communication cost of distributed SGT,

It is known that Serialization Graph Testing (SGT) brings higher concurrency of transactions than other scheduling algorithms. In distributed DBSs, however, the SGT method needs much intersite communication for checking acyclicity of the SG, because the SG has global structure. This communication cost seriously affects the performance of DBSs. This is the major difficulty in using SGT for distributed DBSs.

In distributed DBSs which we are interested in, most of transactions are local transactions. Performance of such DBSs is improved by suppressing communication cost required for scheduling of local transactions. We considered a method for traverse of the SG in which no communications are required for most local transactions.

For distributed DBSs, there are two types of scheduling: *centralized scheduling* using only one scheduler for the whole DBS, and *distributed scheduling* using local schedulers at all sites. Since centralized scheduling harms durability and site autonomy of distributed DBSs, we adopted distributed scheduling. A simple way to implement distributed SGT scheduling is that each site has a copy of the global SG. In this case, however, frequent broadcasts are necessary for synchronization of all SG copies. This means that large communication cost is required for maintenance of the SG. Hence we have chosen another type of distributed SGT scheduler, that is, each site maintains a local subgraph of SG which is concerned with its local data items. In this case, there exists a problem how to detect global cycles, i.e., cycles which do not appear in any local subgraphs.

The detection of global cycle in the distributed SGT scheduling is similar to the distributed deadlock detection using *wait-for graph* (WFG). Distributed deadlock detection using WFG is studied by many researchers and a lot of methods have been proposed [10, 15, 17, 18, 22, 23, 28, 29]. In deadlock detection, a cycle of WFG corresponds to a deadlock. That is, distributed deadlock detection is cycle detection of the distributed WFG. However, there is an important difference between the distributed SGT scheduling and the distributed deadlock detection. In the distributed deadlock detection, any transaction which is not involved in deadlocks can be executed and committed regardless of the progress of the deadlock detection. On the other hand, in the SGT scheduling, the SG is traversed to determine whether to execute an operation. That is, any operation cannot be executed until its SG traverse completes. Hence the completion of the check of the SG should be informed to related sites as soon as possible. In the distributed SGT scheduling, it is difficult to detect the completion of checking of the SG which is executed in a distributed manner. For this purpose, we introduced *fractional tags*. In this chapter, we propose the method for detecting such global cycles using fractional tags. We call our method *Fractional Tag method* (FT). Under FT, each site has to maintain only its local SG and therefore the communication cost for maintaining the SG is suppressed.

We show the correctness of FT and evaluate its performance by simulations in terms of the number of messages which are needed for scheduling.

3.2. Distributed Serialization Graph Testing

The distributed version of the SGT algorithm in section 2.4.6 is presented in this section. The problem is how to maintain the SG in distributed DBSs.

3.2.1 Distributed Scheduling

For distributed DBSs, there are two types of scheduling: *centralized scheduling* using only one scheduler for the whole DBS, and *distributed scheduling* using local schedulers at all sites. In the case of centralized scheduling, all operations are scheduled in the central scheduler. This damages the site autonomy of distributed DBSs. Moreover, centralized scheduling is less durable than distributed scheduling. In distributed DBSs, if one site crashes, other sites are still available except that they cannot access the crashed site. In centralized scheduling, however, this is not always true. When the central scheduler crashes, the whole database stops and even local transactions cannot access local data items. In this way, centralized scheduling damages durability of distributed DBSs. For these reasons, we consider that central scheduling is not suitable for distributed DBSs. Therefore, distributed scheduling is discussed in this thesis.

When SGT is implemented as distributed scheduling, there are two ways of implementation. A simple way is that each site has a copy of the global SG. In this case, maintenance of the SG costs too much. Each site should inform all sites of all changes of the SG it maintains. That is, a site should broadcast a message to all other sites even when the site executes a local transaction. This means that the communication cost grows very much. Therefore, we have chosen another way. In the distributed SGT we propose, each site maintains a subgraph of the SG reflecting only conflicts on the data items which are stored in the site. The subgraph of SG is called *local serialization graph* (local SG).

Definition 3.1 For a history H and a site s, the *local serialization graph* $LSG(s,H) = \langle V, E\rangle$ is the subgraph of $SG(H)$ such that

$V = \{T_i \mid T_i \text{ is in } SG(H) \text{ and } T_i \text{ accessed a data item stored in } s \text{ in } H\}$

$E = \{(T_i, T_j) \mid$ there exists two operations $o_i \in T_i$ and $o_j \in T_j$, which conflict for a data item stored in s, such that $o_i <_H o_j$ where $T_i, T_j \in V\}$ □

In the scheduling algorithm we propose, $LSG(s)$ stores $LSG(s,H)$ where H is the history produced so far.

3.2.2 Node Storing Site Group

In our distributed SGT, each site maintains a local SG which contains all nodes for transactions that conflict with each other for data items stored in the site. *Node storing site group* is defined as follows.

Definition 3.2 For a history H and a transaction T, the *node storing site group* $NS(T,H)$ is a group of all sites whose local SG has the node for transaction T in H. □

In an execution of a *Read* or *Write* operation of a transaction T, every edge added to the SG is incoming to T or outgoing from T. Therefore, all sites whose local SGs are changed by a *Read* or *Write* operation of T are included in $NS(T,H)$.

In the scheduling algorithm we propose, $NS(T,H)$ is maintained by each scheduler in $NS(T,H)$. Therefore, $NS_s(T)$ stores $NS(T,H)$ in site s where H is the history produced so far. $NS_s(T)$ can be maintained locally at s because each site knows where each data item is stored (as described in section 2.5). When s knows that T accesses a data item x, a site which stores x is added to $NS_s(T)$.

3.2.3 Addition of Edges to the Serialization Graph

As described in section 2.4.2, a scheduler maintains readsets and writesets for scheduling. In the case of the distributed SGT which we propose, however, each scheduler cannot maintain readsets or writesets defined in section 2.4.2 because each scheduler does not know which transactions accessed data items in other sites. Instead, schedulers maintain local readsets and local writesets.

Definition 3.3 For a transaction T, a history H and a site s,

$\textit{readset-local}(s,T,H) = \{x \mid x \text{ is stored in } s \text{ and } T \text{ has read from } x \text{ in } H\}$

$\textit{writeset-local}(s,T,H) = \{x \mid x \text{ is stored in } s \text{ and } T \text{ has written to } x \text{ in } H\}$ □

In the scheduling algorithm we propose, *readset-local*(s,T) and *writeset-local*(s,T) stores *readset-local*(s,T,H) and *writeset-local*(s,T,H) respectively where H is the history produced so far. When an operation o of a transaction T is executed in s, s sends *EDGE* messages which inform all sites in $NS(T,H)$ that o will be executed. The site s_i which received the message updates *readset-local*(s,T) or *writeset-local*(s,T), and adds edges to its local SG.

3.2.4 Abortion of Transactions

When a transaction is aborted, other transactions which read from the aborted transactions should also be aborted. Therefore, it is required to search the SG and detect such transactions. Since each site does not maintain the global SG, this SG search should be done at any sites whose local SGs include the aborted transaction. Abortion of a transaction is done in the following way. If a cycle is detected in the SG during the validation of T, the home site of T (denoted s) aborts T and sends *ABORTED* messages which inform all sites in $NS(T,H)$ that T is aborted. Each site received the *ABORTED* message deletes T from its local SG and searches transactions which should be aborted. If such a transaction T' is detected, the site sends an *ABORT* message to the home site of T' (denoted s'). Receiving the *ABORT* message, s' aborts T' and sends *ABORTED* messages as above.

3.2.5 Deletion of Unnecessary Nodes

As mentioned in section 2.4.4, the nodes which are unnecessary for scheduling should be deleted from the SG. In the case that each site maintains a local subgraph of the SG, the node deletion is a little difficult. This is because it is impossible to determine whether a node can be deleted by analyzing a local SG. Even if a node has no incoming edges in a local SG, the node may have an incoming edge in another local SG. To determine locally when a node can be deleted, the home site of committed transaction T maintains the set of nodes which should be deleted before T is deleted.

Definition 3.4 For a transaction T,

$before(T, H) = \{T_i \mid \text{an edge } (T_i, T) \text{ exists in } SG(H)\}$

$before\text{-}local(s, T, H) = \{T_i \mid \text{an edge } (T_i, T) \text{ exists in } LSG(s, H)\}$ □

In the scheduling algorithm we propose, $before(T)$ stores $before(T, H)$ and $before\text{-}local(s, T)$ stores $before\text{-}local(s, T, H)$ where H is the history produced so far. $before\text{-}local(s, T)$ is locally maintained in site s. $before(T)$ is maintained in the home site of T as follows. When a transaction T is committed, the home site of T (denoted s) sends *COMMITTED* messages to sites in $NS(T, H)$. A site s_i which received the *COMMITTED* message makes the set $before\text{-}local(s_i, T)$ and sends it back to s. s receives the *before-local*'s and sets $before(T)$ to the sum of them. If $before(T)$ becomes empty, s sends *DELETE* messages to sites in $NS(T, H)$. When a site s_j receives the *DELETE* message, it deletes T from its local SG and removes T from $before(T_k)$'s (for any T_k). If $before(T_k)$ becomes empty, s_j determines that T_k can be deleted and sends *DELETE* messages.

3.2.6 Distributed SGT Algorithm

The distributed SGT algorithm which we propose is shown in Tables 3.1 and 3.2. In the tables, messages are denoted as follows, where o is an operation, T is a transaction and *before-local* is a set of transactions.

- an *EDGE* message $M_{EDGE}(o)$
- a reply for an *EDGE* message $M_{REPLY_E}(o)$
- a *COMMITTED* message $M_{COMMITTED}(T)$
- a reply for a *COMMITTED* message $M_{REPLY_C}(T, before\text{-}local)$
- an *ABORT* message $M_{ABORT}(T)$

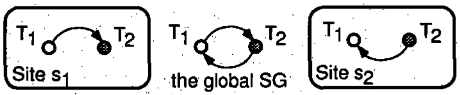

Figure 3.1: An example of a global cycle.

- an *ABORTED* message $M_{ABORTED}(T)$
- a *DELETE* message $M_{DELETE}(T)$

The subtraction of sets is defined as follows

Definition 3.5 For two sets A and B,
$A - B = \{x \mid x \in A \text{ and } x \notin B\}$ □

3.3. Fractional Tag Method

In this section, it is considered how to traverse the SG maintained as described in section 3.2.6. We propose a method which we call Fractional Tag Method to suppress the communication cost for scheduling.

3.3.1 Traverse of Serialization Graph

In the distributed SGT algorithm described in section 3.2.6, each site maintains a local SG. In this case, it is possible that all local SGs are kept acyclic, but the global SG may include a cycle. For example, consider the situation of Figure 3.1. Though local SGs of site s_1 and site s_2 are both acyclic, there is a cycle in the global SG. Such cycles are called *global cycles*. It is a problem how to detect global cycles. We propose a method for traverse of the SG to detect global cycles. Each local SG is traversed only when it is necessary to check whether there is a cycle in the global SG.

In the algorithm described in section 3.2.6, the SG is traversed in function *SGT-validation*(T). The details of the function is described later in section 3.3.4. Roughly speaking, the scheduler using our method traverses the SG as follows.

Suppose that so far a history H has been produced and the scheduler of the site s_1 received an operation o of a transaction T from a transaction manager. If o is a *READ* or *WRITE* operation, then the scheduler sends the *REQUEST* message to all sites in $NS(T, H)$ to request checking whether T will be reachable from T itself in their local SGs after o is executed. The site s_1 is called the *home site* of T.

Table 3.1: The distributed SGT algorithm

The distributed SGT algorithm at site s

```
if an operation $o$ of transaction $T$ is received from the transaction manager (TM) then
    if $o = start$ then
        start $T$
        send a reply that $T$ is started to the TM
    else if $o = commit$ then
        commit $T$
        send $M_{COMMITTED}(T)$ to all sites in $NS_s(T)$
        wait until all $M_{REPLY_C}(before\text{-}local(s_i, T))$ are received
        $before(T) \leftarrow \bigcup_{s_i \in NS_s(T)} before\text{-}local(s_i, T)$
        if $before(T) = \phi$ then
            send $M_{DELETE}(T)$ to all sites in $NS_s(T)$
        send a reply that $T$ is committed to the TM
    if $o = read(x)$ or $o = write(x)$ then
        $NS_s(T) \leftarrow NS_s(T) \cup \{s_j \mid x \text{ is stored in site } s_j\}$
        send $M_{EDGE}(o)$ to all sites in $NS_s(T)$
        wait until all $M_{REPLY_E}(o)$'s are received
        if $SGT\text{-}validation(T) = true$ then
            if $o = read(x)$ then
                read $x$ from the database
                release the readlock of $x$
            if $o = write(x)$ then
                write $x$ to the database
                release the writelock of $x$
            send a reply that $o$ succeeded to the TM
        else /* $SGT\text{-}validation(T) = false$ */
            abort $T$
            send $M_{ABORTED}(T)$ to all sites in $NS_s(T)$
            send a message that $T$ is aborted to the TM
```

Note: *SGT-validation*(T) is a function such that it returns *true* if a serialization graph has no cycles including T and otherwise it returns *false*.

Table 3.2: The distributed SGT algorithm (continued)

A site s_i received a message M from site s does the following:

```
if $M = M_{EDGE}(o)$ then
/* suppose that $o = read(x)$ or $o = write(x)$ */
    $NS_{s_i}(T) \leftarrow NS_{s_i}(T) \cup \{s_j \mid x$ is stored in site $s_j\}$
    if $x$ is stored in $s_i$ then
        if $T \notin LSG(s_i)$ then
            add a node $T$ to $LSG(s_i)$
        if $o = read(x)$ then
            set the readlock of $x$ to $T$
            for each $T_j$ such that $x_i \in writeset\text{-}local(s_i, T_j)$ do
                add an edge $T_j \rightarrow T$ to $LSG(s_i)$
            $readset\text{-}local(s_i, T) \leftarrow readset\text{-}local(s_i, T) \cup \{x\}$
        if $o = write(x)$ then
            set the writelock of $x$ to $T$
            for each $T_j$ such that $x_i \in readset\text{-}local(s_i, T_j) \cup writeset\text{-}local(s_i, T_j)$ do
                add an edge $T_j \rightarrow T$ to $LSG(s_i)$
            $writeset\text{-}local(s_i, T) \leftarrow writeset\text{-}local(s_i, T) \cup \{x\}$
    send $M_{REPLY_E}(o)$ to site $s$
if $M = M_{COMMITTED}(T)$ then
    send $M_{REPLY_C}(before\text{-}local(s_i, T))$ to site $s$
if $M = M_{ABORT}(T)$ then
    abort $T$
    send $M_{ABORTED}(T)$ to all sites in $NS_{s_i}(T)$
    send a message that $T$ is aborted to the TM
if $M = M_{ABORTED}(T)$ then
    delete $T$ from $LSG(s_i)$
    for each $before(T_i)$ maintained in $s_i$ do
        $before(T_i) \leftarrow before(T_i) - \{T\}$
        if $before(T_i) = \phi$ then
            send $M_{DELETE}(T_i)$ to all sites in $NS_{s_i}(T_i)$
    $Abort \leftarrow \{T\}$
    while there exists a transaction $T_i \notin Abort$ which read from $T_j \in Abort$ in site $s_i$ do
        $Abort \leftarrow Abort \cup \{T_i\}$
    for each $T_j \in Abort$ do
        send $M_{ABORT}(T_j)$ to the home site of $T_j$
if $M = M_{DELETE}(T)$ then
    delete $T$ from $LSG(s_i)$
    for each $before(T_i)$ maintained in $s_i$ do
        $before(T_i) \leftarrow before(T_i) - \{T\}$
        if $before(T_i) = \phi$ then
            send $M_{DELETE}(T_i)$ to all sites in $NS_{s_i}(T_i)$
```

When the *REQUEST* message from s_1 is received by a site s_2, s_2 traverses its local SG from node T. If T is reachable from T, then s_2 sends a *CYCLE* message to the home site of T (site s_1) that there is a cycle in the global SG and the traverse ends. Otherwise for each node T_i which is reachable from node T, s_2 sends *REQUEST* messages to all sites in $NS(T_i, H)$ (except s_2 itself) to request checking whether T will be reachable from T_i (not T) in their local SGs. If there are no sites to send *REQUEST* messages, then s_2 sends an *END* message to the home site to inform that s_2 sent no *REQUEST* messages.

Each site s_j which received the *REQUEST* message which indicates a request to check whether T is reachable from T_i, traverses its local SG from T_i. Since T_i is reachable from T in s_2, if T is reachable from T_i, then T is reachable from T. In this case, s_j sends a *CYCLE* message to the home site and the traverse ends with detection of a cycle in the global SG. Otherwise s_j sends *REQUEST* messages or an *END* message as s_2 does.

In this way, *REQUEST* messages are sent in turn. When all sites cannot send *REQUEST* messages, the traverse completes and it is guaranteed that the operation o doesn't cause any cycle in the global SG.

Figure 3.2 shows an example of the SG traverse. In this example, an SG traverse for an operation of the transaction T_1 is executed. The home site s_1 sends a *REQUEST* message to s_2 because $NS(T_1, H)$ shows that there is the node T_1 in the site s_2. s_2 receives the message and traverses its local SG. T_3 is reachable from T_1 in the local SG. According to $NS(T_3, H)$, s_2 sends other *REQUEST* messages to s_3 and s_4. Receiving them, s_3 and s_4 traverse their local SGs. There are no reachable nodes from T_3 in s_3's local SG, then s_3 sends an *END* message to s_1. On the other hand, s_4 sends other *REQUEST* message to s_2 because T_4 is reachable from T_3 in s_4's local SG and $s_2 \in NS(T_4, H)$. s_2 receives the message from s_4 and traverse its local SG. Now suppose that the edge from T_5 to T_1 (broken line in Figure 3.2) does not exist. In this case, only T_5 is reachable from T_4 in the local SG. However, T_5 causes no *REQUEST* messages because $NS(T_5, H)$ includes s_2 only. Therefore, s_2 sends an *END* message to s_1. In the case that the edge from T_5 to T_1 exists, T_1 is reachable from T_4 in the local SG. This means that there is a cycle including T_1 in the global SG. Therefore, s_2 sends a *CYCLE* message to s_1.

3.3.2 Fractional Tags

In the method mentioned in section 3.3.1, the SG traverse completes when all sites cannot send *REQUEST* messages. Now it is a problem how to notify the

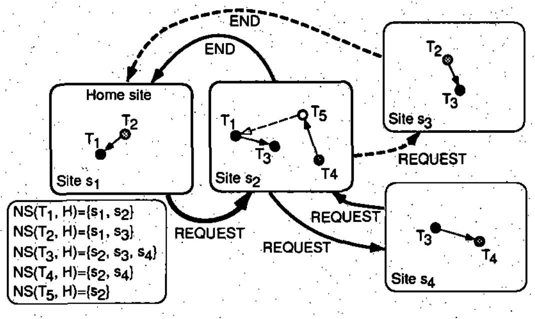

Figure 3.2: An example of SG traverse.

home site of the completion of the traverse. It should be done as soon as possible because any operation cannot be executed until its validation, i.e. SG traverse, completes. For this purpose, *fractional tags* are introduced. A fractional tag is a fraction which is attached in a message.

In order to notify when traverse of the SG is completed, fractional tags are used as follows. Suppose s_1 is the home site of T and so far a history H has been produced. When the function *SGT-validation*(T) is called, s_1 sends *REQUEST* messages to all sites in $NS(T, H)$ with the fractional tag $1/|NS(T, H)|$.

Receiving a *REQUEST* message with the fractional tag t, the site s_2 traverses its local SG. If there are no sites to send *REQUEST* messages, then s_2 sends an *END* message to the home site with the tag t which is equal to the received tag. On the other hand, if there are m sites to send *REQUEST* messages, then s_2 sends the messages to them with the fractional tag t/m.

The home site stores the sum of the values of the tags of the received *END* messages for each operation. If the sum amounts to 1, then the site concludes that no more *REQUEST* messages of the operation will be sent from any site. The correctness of this algorithm is shown in section 3.4.

Figure 3.3 shows an example of transfer of fractional tags. In this example, the site s_1 passes the tag $1/2$ to the sites s_2 and s_3. Then s_2 returns the received tag $1/2$ to s_1. On the other hand, s_3 divides the received tag into three and sends $1/6$ to three sites. Three sites received them returns the tag $1/6$ to s_1. The sum of tags received by s_1 is $1/2 + 1/6 + 1/6 + 1/6 = 1$. Clearly, the numerator of a fractional tag is always 1. Therefore, a fractional tag can be implemented by only one integer which represents denominator.

3.3.3 Unnecessary Messages

Consider the situation shown in Figure 3.4. In the traverse for an operation of T_1, the site s_1 sends *REQUEST* messages to the site s_2 and s_3 because T_2 is

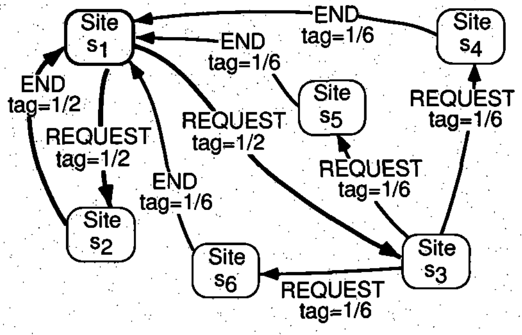

Figure 3.3: An example of fractional tags.

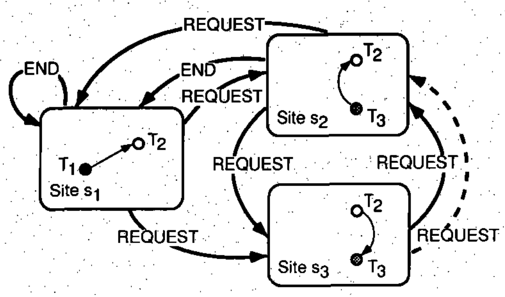

Figure 3.4: An example of an useless message.

reachable from T_1. Receiving the message, s_3 sends a *REQUEST* message to s_2 because T_3 is reachable from T_2. Receiving the message from s_3, s_2 sends *REQUEST* messages to s_1 and s_3 because T_2 is reachable from T_3. Then s_3 sends a *REQUEST* message to s_2 again. This message is useless because the same message has been sent before. To avoid such a case, each *REQUEST* message includes the set of nodes which have already been traversed. We call the set *track*. Even when T_j is reachable from T in the local SG, if T_j is included in the track of the received *REQUEST* message, *REQUEST* messages are not sent.

3.3.4 Algorithm

The proposed method is called *Fractional Tag Method* (FT). The function *SGT-validation* using FT for a transaction T (denoted $SGT\text{-}validation_{FT}(T)$) is shown in Table 3.3.

Following messages are used, where o is an operation, T and T' is a transaction, t is a fractional tag and Trk is a set of nodes mentioned in section 3.3.3.

- a *REQUEST* message $M_{REQUEST}(T, T', t, Trk)$
- an *END* message $M_{END}(T, t)$

Table 3.3: A function *SGT-validation*(T) using FT

A function *SGT-validation*$_{FT}(T)$ at site s

```
$tg \leftarrow 1/|NS_s(T)|$
$Trk \leftarrow \{T\}$
$sum \leftarrow 0$
send $M_{REQUEST}(T, T, tg, Trk)$ to all sites in $NS_s(T)$
while $sum < 1$ do
    wait for a message $M$ from other site
    if $M = M_{END}(T, tg)$ then
        $sum \leftarrow sum + tg$
    if $M = M_{CYCLE}(T)$ then
        return $false$
return $true$
```

A site s_i received $M_{REQUEST}(T, T_{cur}, tg, Trk)$ does the following:

```
if $T_{cur}$ has no outgoing edges in $LSG(s_i)$ then
    send $M_{END}(T, tg)$ to $T$'s home site
else if $T$ is reachable from $T_{cur}$ in $LSG(s_i)$ then
    send $M_{CYCLE}(T)$ to $T$'s home site
else
    for each $T_i \in R(T_{cur})$ do
  $(R(T_{cur}) = \{T \mid T$ is reachable from $T_{cur}$ in $LSG(s_i)$ and $T \notin Trk\})$
        $tg_i \leftarrow (tg/|R(T_{cur})|)/(|NS_{s_i}(T_i)| - 1)$
        $Trk_i \leftarrow Trk \cup R(T_{cur})$
        send $M_{REQUEST}(T, T_i, tg_i, Trk_i)$ to all sites in $NS_{s_i}(T_i)$ except $s_i$
```

- a *CYCLE* message $M_{CYCLE}(T, t)$

$LSG(s)$ and $NS_s(T)$ are the same as section 3.2.6.

3.4. Correctness

3.4.1 Correctness Proof

In this section, we show that SGT using FT causes no deadlocks.

Definition 3.6 A message M is *held* by site s iff s received M and the process for M has not completed yet. □

Lemma 3.1 In the execution of function *SGT-validation*$_{FT}(T)$, the sum of the following values is always 1 unless the function returns $false$.

- fractional tags in all *REQUEST* messages of T held by sites or on the communication links.
- fractional tags in all *END* messages of T received by the home site of T or on the communication links.

If the function returns $false$, the sum must be less than 1.
[Proof] When *SGT-validation*$_{FT}(T)$ is called, the home site sends the *REQUEST* messages with the tag $1/|NS(T)|$ to all sites in $NS(T)$ (line 4). At this time, the sum of the tags of all *REQUEST* messages is 1.

Suppose that a site s received a *REQUEST* message of o with tag tg. Three cases are possible.

1. s sends an *END* message to the home site, s sends the message with the tag which is equal to the tag of the received *REQUEST* message (line 13). Therefore, the sum is not changed by the processing for the *REQUEST* message.

2. s sends *REQUEST* messages to the other sites. The sum of the tags of these *REQUEST* messages is equal to tg (lines 18 and 20). That is, the sum is not changed by the processing for the *REQUEST* message in this case too.

3. s sends *CYCLE* messages to T's home site. The tag received by s is not sent and it is lost (line 15). Thereby the sum must decrease. This is the only case that *SGT-validation*$_{FT}(T)$ returns $false$.

Therefore, the sum is 1 unless function *SGT-validation*$_{FT}(T)$ returns $false$. and if the function returns $false$, the sum must be less than 1. □

Lemma 3.2 When *SGT-validation*$_{FT}(T)$ is called, *REQUEST* messages must stop to be sent eventually.
[Proof] At first, the number of *REQUEST* messages sent from T's home site is bounded by the number of sites (line 4). When the site s received a *REQUEST* message with a node T_{cur} and a track Trk, s traverses all nodes those are reachable from T_{cur} in its local SG. For each $T_i \notin Trk$, which is reachable from T_{cur}, s adds T_i to Trk (line 19) and passes *REQUEST* messages including the new track (line 20). Therefore, this new track included in the passed messages is larger than that of the received message. Since the number of nodes in the track is bounded by the number of nodes in the global SG, this message passing must end eventually. □

Lemma 3.3 In *SGT-validation*$_{FT}(T)$, the sum of the fractional tags in *END* messages of T received by the home site amounts to 1 iff *REQUEST* messages of T stop to be sent unless *SGT-validation*$_{FT}(T)$ returns $false$.

[Proof] (if) Suppose that *REQUEST* messages of T stopped to be sent without *SGT-validation*$_{FT}(T)$ returning $false$. *REQUEST* messages of T will disappear eventually. Therefore, by Lemma 3.1, the sum of tags of *END* messages of T becomes 1 and all *END* messages of T should arrive at the home site of T.

(only if) Suppose that the sum of the fractional tags in *END* messages of T received by the home site amounted to 1. Every *REQUEST* message or *END* message has a positive fractional tag, it follows from Lemma 3.1 that no *REQUEST* messages are not held by any site or on the communication link. Therefore, if the sum amounts to 1, then *REQUEST* messages of T has stopped to be sent. By Lemma 3.1, *SGT-validation*$_{FT}(T)$ does not return $false$. □

Lemma 3.4 Under SGT using FT, any operation o received from the transaction manager must be executed eventually unless T is aborted.

[Proof] When operation o is received from the transaction manager, the scheduler calls function *SGT-validation*$_{FT}$. In the function, a *REQUEST* message is sent to n sites with the tag $1/n$ where $n = |NS(T)|$ (line 5). Suppose that T is not aborted. This means that *SGT-validation* did not return $false$. By Lemma 3.2, *REQUEST* messages must stop to be sent eventually. It follows from Lemma 3.3 that the sum of the tags in *END* messages of T received by the home site must amount to 1. Then the function *SGT-validation*$_FT$ returns *true* and o is executed. Therefore, o must be executed eventually unless T is aborted. □

Lemma 3.5 Under SGT using FT, every active transaction must be committed or aborted eventually.

[Proof] By Lemma 3.4, every operation of a transaction T sent to scheduler must be executed unless T is aborted. Therefore, if a transaction T is not aborted, T must be committed eventually, otherwise it is aborted. □

Theorem 3.1 No deadlocks occur under SGT using FT.

[Proof] From Lemma 3.5, no transactions stop infinitely. Hence no deadlocks occur. □

3.4.2 Livelocks

As described before, SGT using FT causes no deadlocks. However, SGT (whether using FT or not) may cause livelocks, that is, there may be some transactions which repeat restart infinitely. When there is a cycle in the SG, some transactions in the cycle are aborted, then the cycle disappears. Livelocks may occur when all transactions in the cycle are restarted by abortion, and the same cycle is produced by restarted transactions. Therefore, the livelocks can be avoided by selecting one transaction in a cycle and prevent it from abortion. This can be easily implemented by modifying basic SGT algorithm.

3.5. Evaluation

FT is designed to suppress the communication cost of the distributed SGT scheduling when most transactions access only local data items. In this section, we evaluate the communication cost of SGT using FT (denoted SGT-FT) and compare it with that of the simple implementation of distributed SGT, i.e., all sites maintain a global copy of the SG. Therefore, we call it *SGT with Global Copy* (SGT-GC). SGT-GC is shown in Tables 3.4 and 3.5. In SGT-GC, each site s maintains $SG(H)$, $readset(T,H)$, $writeset(T,H)$ and $before(T,H)$ (for any T) locally. In Table 3.5, they are denoted by SG_s, $readset_s(T)$, $writeset_s(T)$ and $before_s(T)$ respectively.

In this section, n denotes the number of sites in the database system and $length(T)$ denotes the number of operations in transaction T.

3.5.1 Comparison in Communication Cost

The communication cost depends on the number of messages needed for scheduling transactions. In this section, the number of messages needed for scheduling of SGT-FT is compared with that of SGT-GC. Since messages sent from a site to itself do not require any communication, such messages are not counted.

In the case of SGT-FT, the number of messages differs much among transactions. Suppose that so far a history H has been produced. If no cycles are detected, then the number of messages for scheduling one operation is

$$\begin{aligned} MsgOpr_{FT}(T,H) &= 2\cdot(|NS(T,H)|-1) + \sum_{T_i\in R(T,H)} (|NS(T_i,H)|-1) \\ &+ \sum_{T_i\in RE(T,H)} (|NS(T_i,H)|-1) \end{aligned}$$

where

$$\begin{aligned} R(T,H) &= \{T\}\cup\{T_i \mid T_i \text{ is reachable from } T \text{ in } SG(H)\} \\ RE(T,H) &= \{T_i \mid T_i \in R(T,H) \text{ and } T_i \text{ has no outgoing edges in } SG(H)\}. \end{aligned}$$

Table 3.4: The SGT-GC algorithm

The SGT-GC algorithm at site s

```
**if** an operation $o$ of transaction $T$ is received from the transaction manager (TM) **then**
    **if** $o = start$ **then**
        **start** $T$
        **send** a reply that $T$ is started **to** the TM
    **else if** $o = commit$ **then**
        **commit** $T$
        **send** $M_{COMMITTED}(T)$ **to** all sites
        **send** a reply that $T$ is committed **to** the TM
    **if** $o = read(x)$ or $o = write(x)$ **then**
        **send** $M_{EDGE}(o)$ **to** all sites
        **if** $T$ is reachable from $T$ in $SG_s$ **then**
        /* $SG_s$ has no cycles including $T$ */
            **if** $o = read(x)$ **then**
                **read** $x$ from the database
                **release** the readlock of $x$
            **if** $o = write(x)$ **then**
                **write** $x$ to the database
                **release** the writelock of $x$
            **send** a reply that $o$ succeeded **to** the TM
        **else** /* $SG_s$ has a cycle including $T$ */
            **abort** $T$
            **send** $M_{ABORTED}(T)$ **to** all sites
            **send** a message that $T$ is aborted **to** the TM
```

Table 3.5: The SGT-GC algorithm (continued)

A site s_i received a message M from site s does the following:

```
if $M = M_{EDGE}(o)$ then
    /* suppose that $o = read(x)$ or $o = write(x)$ */
    if $x$ is stored in $s_i$ then
        if $o = read(x)$ then
            set the readlock of $x$ to $T$
        if $o = write(x)$ then
            set the writelock of $x$ to $T$
    if $T \notin SG_{s_i}$ then
        add a node $T$ to $SG_{s_i}$
    if $o = read(x)$ then
        for each $T_j$ such that $x_i \in writeset_{s_i}(T_j)$ do
            add an edge $T_j \rightarrow T$ to $SG_{s_i}$
        $readset_{s_i}(T) \leftarrow readset_{s_i}(T) \cup \{x\}$
    if $o = write(x)$ then
        for each $T_j$ such that $x_i \in readset_{s_i}(T_j) \cup writeset_{s_i}(T_j)$ do
            add an edge $T_j \rightarrow T$ to $SG_{s_i}$
        $writeset_{s_i}(T) \leftarrow writeset_{s_i}(T) \cup \{x\}$
if $M = M_{COMMITTED}(T)$ then
    if $before_{s_i}(T) = \phi$ then
        delete $T$ from $SG_{s_i}$
        for each $before_{s_i}(T_i)$ maintained in $s_i$ do
            $before_{s_i}(T_i) \leftarrow before_{s_i}(T_i) - \{T\}$
            if $before_{s_i}(T_i) = \phi$ then
                delete $T_i$ from $SG_{s_i}$
if $M = M_{ABORT}(T)$ then
    abort $T$
    send $M_{ABORTED}(T)$ to all sites
    send a message that $T$ is aborted to the TM
if $M = M_{ABORTED}(T)$ then
    delete $T$ from $SG_{s_i}$
    for each $before_{s_i}(T_i)$ do
        $before_{s_i}(T_i) \leftarrow before_{s_i}(T_i) - \{T\}$
        if $before_{s_i}(T_i) = \phi$ then
            delete $T_i$ from $SG_{s_i}$
    $Abort \leftarrow \{T\}$
    while there exists a transaction $T_i \notin Abort$ which read from $T_j \in Abort$ in site $s_i$ do
        $Abort \leftarrow Abort \cup \{T_i\}$
    for each $T_j \in Abort$ do
        send $M_{ABORT}(T_j)$ to the home site of $T_j$
```

The first term of $MsgOpr_{FT}(T, H)$ is the number of *EDGE* messages and their replies. The term "-1" means the exception of a message send to itself. The second term is the number of *REQUEST* messages. The third term is the number of *END* messages. Consider a site s which received a *REQUEST* message of an operation of transaction T. s traverses its local SG and sends *REQUEST* messages or an *END* message. When the node from which the traverse starts, say T_i, has an outgoing edge, *REQUEST* messages are sent. The number of *REQUEST* messages which are sent at this time is

$$\sum_{T_j \in RL(T,H,s)} (|NS(T_j, H)| - 1)$$

where

$$RL(T, H, s) \;=\; \{T_i \mid T_i \text{ is reachable from } T \text{ in the local SG of } s\}.$$

The term "-1" means that a *REQUEST* message is not sent to s itself. The second term of $MsgOpr_{FT}(T, H)$ is derived from the fact that $\bigcup_s RL(T, H, s) = R(T, H)$. An *END* message is sent when T_i is in $RE(T, H)$, that is, T_i has no outgoing edge. Therefore, the third term of $MsgOpr_{FT}(T, H)$ is obtained. If there is a node which has two or more paths from T in the SG, the number of messages is more than $MsgOpr_{FT}(T, H)$ because the node is traversed repeatedly. This extra messages can be removed by a little modification of the algorithm.

Definition 3.7 A transaction T is *strictly-local* iff T and all transactions reachable from T in the SG access only the local data items. □

The number of messages which SGT-FT needs for scheduling a transaction T (when T is not aborted and restarted) is

$$MsgTrn_{FT}(T, H) = length(T) \cdot MsgOpr_{FT}(T, H) + 3 \cdot (|NS(T, H)| - 1)$$

The second term of $MsgTrn_{FT}(T, H)$ is the number of *COMMITTED* messages, their replies and *DELETE* messages.

In the case of SGT-GC, each site maintains the global SG. Therefore, *EDGE* messages are always sent to all sites. However, no messages are required for traverse of the SG. Moreover, node deletion can be done locally. The number of messages which SGT-GC needs for scheduling a transaction T (when T is not aborted and restarted) is

$$MsgTrn_{GC}(T, H) = length(T) \cdot (n - 1) + (n - 1)$$

The term "-1" in both $(n-1)$'s means the exception of a message send to itself. The first term is the number of *EDGE* messages. EDGE messages are sent for each operation. The second term is the number of *COMMITTED* messages. They are also sent to all sites.

$MsgTrn_{FT}(T, H)$ may exceed $MsgTrn_{GC}(T, H)$ when T and reachable transactions accesses data items in many sites. However, if T is strictly-local, then $MsgTrn_{FT}(T, H) = 0$ because $|NS(T, H)| = 1$ and $|NS(T_i, H)| = 1$ for any $T_i \in R(T, H)$. That is, T requires no communication for scheduling. If many transactions are strictly-local, SGT-FT is useful in suppressing the communication cost of the distributed SGT.

3.5.2 Overview of Simulations

To examine quantitatively usefulness of SGT-FT in comparison with SGT-GC, we execute simulations. The number of messages required for scheduling is evaluated.

For simplicity, a transaction is regarded as a string of operations in the experiment.

Locality of data access is the most important factor that affects the performance of SGT-FT. *Locality* is defined as the ratio of local transactions in all executed transactions.

It is assumed that there are 10 sites in the distributed database system. Moreover, we have following assumptions:

1. 100 data items are stored in one site.

2. The transaction size is fixed; that is, all transactions contain 8 read/write operations.

3. 25% of all operations are write operations.

4. Each global transaction accesses at most 3 sites.

5. All messages are received in the order they are sent.

The SGT-GC scheduler which we implemented does not wait for replies of the broadcasted messages. Without the assumption 5, SGT-GC needs reply messages to make sure that broadcasted messages are received. It causes much increase of the number of messages in SGT-GC. Therefore, the assumption 5 is in favor of SGT-GC.

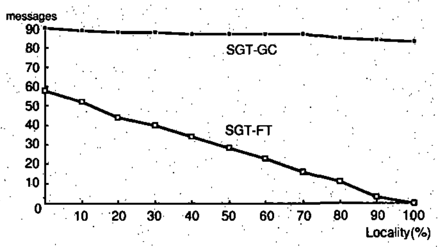

Figure 3.5: Average number of messages.

We introduced several types of messages in section 3.3.4 for SGT-FT. In implementation, additional types of messages are used to abort transactions and delete unnecessary nodes from the SG.

3.5.3 Simulation Results

We compared SGT-FT with SGT-GC in terms of the number of messages needed for scheduling. The experiments investigated the effects of variations in data locality of transactions.

Figure 3.5 depicts the average number of messages which was needed to execute one transaction. The average number of messages under SGT-FT is less than that under SGT-GC. Under SGT-FT, the number of messages decreases as locality grows, while it is almost fixed under SGT-GC. This result shows that SGT-FT is much more efficient than SGT-GC when locality is high.

For each transaction, we measured the number of messages which were sent for scheduling the transaction. Figure 3.6 depicts the distribution of these numbers of 100 transactions in low locality (locality = 20%). Under SGT-GC, most transactions needs 80 $\sim$ 90 messages, Under SGT-FT, the numbers of messages are widely distributed. Figure 3.7 depicts the same distribution in high locality (locality = 80%). Under SGT-GC, the distribution is almost the same as that in Figure 3.6. Under SGT-FT, it is concentrating in the range of 0 $\sim$ 10 messages This shows that most local transactions need few messages in SGT-FT.

3.6. Conclusions

In this chapter, a scheme for maintenance and traverse of the SG in distributed DBSs was proposed to suppress the communication cost of SGT.

In order to suppress the communication cost of maintenance the SG, each site maintains only a local subgraph of the SG which is concerned with its local data items. However, acyclicity of the global SG cannot be examined

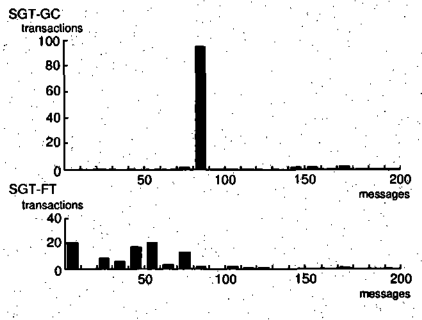

Figure 3.6: The distribution of numbers of messages (locality = 20%).

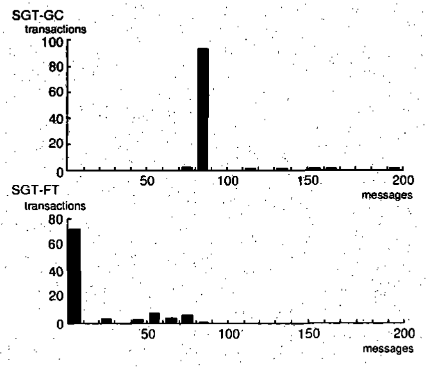

Figure 3.7: The distribution of numbers of messages (locality = 80%).

only by analyzing a local subgraph of the SG. It is too costly to compose the global SG to check acyclicity. In the proposed scheme, the global SG is not composed. A traverse of the SG is performed through message passing among sites which maintain local SGs required to check acyclicity of the global SG. Since every sites are not always involved in the traverse, communication cost can be suppressed. Especially, most local transactions are scheduled without any communications. Moreover, several messages can be sent to different sites simultaneously. Therefore, a traverse of the SG may be executed in parallel at several sites. The problem is how to detect the completion of a traverse of the SG to related sites as soon as possible. This is difficult because a traverse of the SG is executed at several sites and sites involved the traverse are determined dynamically. For this purpose, fractional tags are introduced. A fractional tag is a fraction which is attached to each message. The site which started an SG traverse can realize that the traverse is completed, by confirming that the sum of fractional tags attached to messages it received is 1.

The performance of SGT with FT is compared with that of the simple SGT in which each site maintains the global SG by simulations. By the simulation results, it is corroborated that the effect of FT is large for the databases in which local transactions are dominant.

Chapter 4

A Scheduling Algorithm for Suppressing Scheduling Cost

4.1. Introduction

In this chapter, a variant of SGT is presented. It is proposed to suppress the effect of the scheduling cost to the system performance.

Using FT scheme described in chapter 3, the communication cost for scheduling can be suppressed for local transactions. Nevertheless, the communication cost is still large for global transactions. In order to alleviate the effect of the scheduling cost, decreasing the number of validations is useful. Only one validation is done for each transaction in SGT certification, while a validation is required for each operation in SGT. This is why we adopted SGT certification. Under SGT certification, a scheduler traverses the SG only once for each transaction. Thus the scheduling cost of SGT is suppressed. The drawback in SGT certification is that the abortion of a transaction is delayed until the transaction is about to commit. This delay of abortion is undesirable because it increases the waste of execution time. To make matters worse, it also increases the possibility of cascading aborts. Cascading aborts brings more abortions and the abortions may cause another abortions. Therefore, it is significant for SGT certification how to avoid such cascading aborts. One of certifications, called OCC [19], gave good suggestions. Unlike other certifications, no cascading aborts occur under OCC. It is the feature of OCC that it first performs write operations to internal buffers, and the values are then written to the actual database at the termination of the transaction that wrote them. Since the values in the buffers cannot be accessed by other transactions, write operations are deferred practically. We considered applying this feature of OCC to SGT certification and propose a variant of SGT. In our algorithm, substantial write operations are deferred as in OCC. Therefore, we call it *SGT with Write Deferring* (SGT-WD). The merit of SGT-WD is that it suppresses the number of abortions despite of its fewer validations than that of SGT. The fewer validations means the smaller effect of scheduling cost to system performance in distributed SGT. Moreover, it is the feature of SGT-WD that data items which is written by a transaction T are actually updated only when T is sure to be committed. Thus data restoring is unnecessary when transaction is aborted. This is another large merit of SGT-WD.

In this chapter, we present the SGT-WD algorithm and show its correctness. We consider that its merits are apparent especially in distributed database systems that need communications for scheduling. Therefore, we evaluate the performance of SGT-WD, SGT, and SGT certification by simulations on distributed database systems.

4.2. Certifications

Under usual scheduling algorithms, every time a scheduler receives an operation, the scheduler does validation and decides whether to accept, reject, or delay the operation. Under Certifications, a scheduler can immediately schedule each operation it receives. When the transaction is about to commit, it does validation for the whole transaction. If it is concluded from the validation that all is well, then the transaction is committed. If it is detected that it has inappropriately scheduled conflicting operations, then the transaction should be aborted. In this way, operations are aggressively scheduled under Certifications, in the hope that no conflicts will occur. Therefore, the processing time of transactions is shorter than in other scheduling algorithms.

On the other hand, operations are scheduled even if they cause loss of consistency; this is not detected until validation, which is done at the end of the transaction. Therefore, the abortion of a transaction tends to be delayed. This delay of abortion is undesirable by two reasons. First, when a transaction is aborted, the execution time of the transaction is wasted. Delay of abortion increases this wasted time. Second, if a transaction T which should be aborted continues its execution, other transaction T' may read from T before T is aborted. When T is aborted, T' should also be aborted, that is, cascading abort occurs. In this way, delay of abortion increases the possibility of cascading aborts.

Several types of Certifications has been proposed. They differs in the way of validation. Locking-based Certification [14] uses *optimistic locks* which works as locks used in 2PL. Timestamp-based Certification [5][7][14][28][32] uses timestamps for validation. Certification which uses SGT for validation has also been proposed [4][6][32]. We call it *SGT certification.* In order to suppress the effect of the cost for traverse of the SG, we developed an algorithm based SGT certification. Among drawbacks of Certification approach, we considered that increase of cascading aborts is most serious. Cascading aborts bring more abortions and they may cause another abortions. Such a cascade of abortions seems to affect the throughput of DBSs seriously. Therefore, it is studied how to improve SGT certification in order to avoid such cascading aborts.

4.3. Optimistic Concurrency Control

One of Certifications, called *Optimistic Concurrency Control* (OCC) [19], gave good suggestions for improvement of SGT certification. Like other Certifications, an OCC scheduler aggressively schedules operations. The unique feature of OCC is that it defers substantial write operations by using internal buffers. In OCC, an execution of a transaction T is divided into the following three phases:

Read phase: In this phase, all read operations are executed immediately, and are completely unrestricted. All write operations take place in internal buffers that cannot be accessed by other transactions.

Validation phase: In this phase, validation is performed to determine whether the changes made by T will cause inconsistency in the database. If not, the validation is successful; otherwise, it fails.

Write phase: In this phase, the values in internal buffers are written into the actual database. At this time, the modification made by T become effective. T commits at the end of the phase.

A transaction T first enters the read phase. When T is about to execute a commit operation, it enters the validation phase. If the validation succeeds, T enters the write phase and is committed. Otherwise, T is aborted and restarted.

In OCC, in order to verify that serializability is preserved, the scheduler explicitly assigns each transaction a unique integer called *transaction number* $t(i)$ during the course of its execution. The meaning of transaction numbers in validations is as follows: there must exist a serially equivalent execution in which transaction T_i comes before transaction T_j whenever $t(i) < t(j)$. Transaction numbers are assigned at the end of the read phase. In the validation phase, the validation condition is checked. *readset*(T) and *writeset*(T) are defined as in section 2.4.2. For each transaction T_j with transaction number $t(j)$, and for all T_i with $t(i) < t(j)$; one of the following three conditions must hold.

1. T_i completes its write phase before T_j starts its read phase.

2. *writeset*$(T_i) \cap$ *readset*(T_j) is empty and T_i completes its write phase before T_j starts its write phase.

3. *writeset*$(T_i) \cap ($*readset*$(T_j) \cup$ *writeset*$(T_j))$ is empty and T_i completes its read phase before T_j completes its read phase.

If none of above three conditions hold for some T_i, the validation of T_j fails. For more details, see Kung and Robinson[19].

OCC checks the consistency according to overlapping of concurrent executions. The concurrency of transactions under OCC is not so high as under SGT.

OCC's advantage over other Certifications is that an abortion of a transaction affects no other transactions. Write operations are executed on the actual database only when the transaction is sure to be committed. Therefore, when a transaction aborts, there are no data items modified by the transaction. Accordingly, no more abortions are caused by the abortion. That is, all executions produced by OCC are guaranteed not to cause cascading aborts. We call such executions *cascadeless*. It is also said that the executions *avoid cascading aborts.*

Regarding transaction numbers as a kind of timestamps, the validation of OCC resembles that of Timestamp-based Certification. As described above, however, OCC is more restrictive and therefore concurrency provided by OCC is lower than Timestamp-based Certification.

4.4. Proposed Algorithm

We apply the OCC approach to SGT certification and propose a new algorithm that overcomes the disadvantages of SGT. The important points are as follows:

1. All operations are scheduled immediately and they are validated later. Therefore, the scheduling cost of transactions is smaller than that of SGT.

2. Write operations are deferred until the validation is complete. As a result, executions produced by this algorithm are cascadeless.

We call our algorithm *Serialization Graph Testing with Write Deferring* (SGT-WD). The SGT-WD algorithm is shown in Table 4.1. Like OCC, SGT-WD divides the execution of a transaction into three phases. For simplicity, Table 4.1 shows the algorithm for centralized scheduler. It is easily modified to the distributed version as shown in section 3.2.1.

4.5. Correctness

In this section, we show the correctness of SGT-WD. The correctness of the usual SGT is given by Theorem 2.1. Though SGT-WD also uses the serialization graph for scheduling, the correctness of SGT-WD is not shown by the theorem. Since an SGT-WD scheduler defers write operations of a transaction, the execution order of operations differs from the original transaction. Therefore, Theorem 2.1 does not hold immediately for SGT-WD. The correctness of SGT-WD is shown in this section.

Definition 4.1 For each transaction T_i in H, *a write-deferred transaction* T_i^{wd} is a transaction such that

- T_i^{wd} has the same set of operations as T_i.
- T_i^{wd}'s read operations are executed in the same order as those of T_i.
- T_i^{wd}'s write operations are executed after its last read operation.
- T_i^{wd} writes the same values to data items as T_i.

Note that T_i and T_i^{wd} read and write the same values.

Definition 4.2 For a serial history H_s, *a write-deferred serial history* H_s^{wd} is a history which is derived from H_s by replacing any $T_i \in H_s$ with T_i^{wd}.

Lemma 4.1 A history H_s^{wd} preserves database consistency.
[Proof] Each transaction $T_i^{wd} \in H_s^{wd}$ read and write the same values as corresponding $T_i \in H_s$. From the definition of H_s^{wd}, T_i^{wd} and T_i appear in the same order in each history. Therefore, H_s^{wd} has the same effect on database as H_s does. Since H_s preserves database consistency, H_s^{wd} preserves database consistency. □

Table 4.1: The SGT-WD algorithm

when a transaction T is in the read phase

```
if an operation $o$ of $T$ is received from the transaction manager (TM) then
    if $o = start$ then
        add a node $T$ to $SG$
        start $T$
        send reply such that $T$ is started to the TM
    else if $o = commit$ then
        $T$ enters the validation phase
    else if $o = read(x)$ then
        set the readlock of $x$ to $T$
        for each $T_i$ such that $x \in writeset(T_i)$ do
            add an edge $T_i \rightarrow T$ to $SG$
        add $x$ to $readset(T)$
        write $x$ from the actual database
        release the readlock of $x$
        send reply such that $o$ succeeded to the TM
    else /* $o = write(x)$ */
        store $x$ to an internal buffer
    (do not add edges to the SG here)
        send reply such that $o$ succeeded to the TM
```

when T is in the validation phase

```
for each $x_i$ which is stored in internal buffers by $T$ do
    set the writelock of $x_i$ to $T$
    for each $T_j$ such that $x_i \in readset(T_j) \cup writeset(T_j)$ do
        add an edge $T_j \rightarrow T$ to $SG$
    add $x_i$ to $writeset(T)$
if $SGT\text{-}validation(T, SG) = true$ then
    $T$ enters the write phase
else /* $SGT\text{-}validation(T) = false$ */
    delete the node $T$ from $SG$
    abort $T$
```

when T is in the write phase

```
for each $x_i$ stored in internal buffers by $T$ do
    write $x_i$ to the actual database
    release the writelock of $x_i$
commit $T$
delete unnecessary nodes from $SG$
send reply to the TM such that $o$ succeeded
```

Theorem 4.1 Let H be a history produced by SGT-WD. $SG(H)$ is acyclic iff H preserves database consistency.

[Proof] Table 4.1 shows that although write operations are deferred, the executed transaction writes the same value as the original transaction in SGT-WD. This means that an SGT-WD scheduler executes T_i^{wd} instead of T_i. Therefore, H is an execution in which T_i^{wd}s are executed concurrently. For SGT-WD, Theorem 2.1 says that $SG(H)$ is acyclic iff one element of $PH(H)$ is conflict equivalent to a write-deferred serial history H_s^{wd}. Since H_s^{wd} preserves database consistency from Lemma 4.1, $SG(H)$ is acyclic iff H preserves database consistency. □

As mentioned in section 2.4.5, SGT-WD uses readlocks and writelocks to execute read/write processes without interleaving with other conflicting operations. Therefore, the following theorem is derived immediately.

Theorem 4.2 SG maintained by an SGT-WD scheduler is always equal to $SG(H)$. □

The above two theorems show that an SGT-WD scheduler always produces serializable executions.

4.6. Evaluation

4.6.1 Comparison by Examples

In this section, we compare SGT-WD with OCC and SGT certification. To explain the features of the two algorithms, we show some examples of concurrent executions in figures. In the figures, the following symbols are used:

- S denotes the start of the transaction.
- $R(x)$ ($W(x)$) denotes the read (write) operation on data item x.
- V denotes the validation. It includes the validation phase and the subsequent write phase.
- C denotes the commitment of a transaction.
- A denotes the abortion of a transaction.

Figure 4.1 shows concurrent executions of transactions produced by OCC and SGT-WD respectively. In the case of OCC, since *readset*(T_2) and *writeset*(T_1)

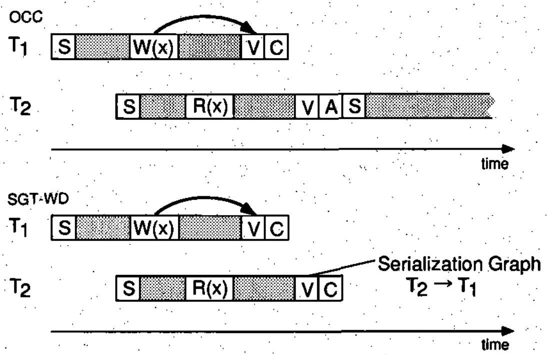

Figure 4.1: Comparison between OCC and SGT-WD

conflict, condition 1 in section 4.3 must hold in the validation of T_2. However, T_2 starts its read phase before T_1 completes its write phase. Therefore, the validation fails and T_2 is aborted and restarted.

In the case of SGT-WD, since conflicts between T_1 and T_2 cause no cycles in the SG, the validation is successful and T_2 is committed. In this situation, the processing time of T_2 under SGT-WD is shorter than that under OCC.

Generally, suppose that *read*(x) of a transaction T_i is scheduled after *write*(x) of a committed transaction T_j. Under OCC, if T_i has been started before T_j was committed, the validation of T_i fails and T_i must be aborted. In the case of SGT-WD, the validation of such T_i succeeds unless T_i writes the same data item. Such a situation happens frequently with database systems in which processing time of transactions tend to become long. This fact shows the advantage of SGT-WD over OCC. The class of executions produced by SGT-WD contains properly than that of OCC because the validation of SGT-WD is based strictly on the definition of serializability, which is a correctness criterion both algorithms use.

Figure 4.2 shows an execution produced by SGT certification and SGT-WD. Suppose that a transaction T_1 is involved in a cycle including other transactions (not T_2) and is aborted. Consider SGT certification first (top of Figure 4.2). A transaction T_1 is aborted by the failure, and then T_2, which read the data item x written by T_1, must also be aborted; that is, a cascading abort occurs. However, since T_2 has already been committed when T_1 is aborted, T_2 cannot be aborted and the database can no longer recover to a consistent state. This shows that SGT certification may produce executions that are not recoverable. To make the execution recoverable, SGT certification should defer the commit of T_2 until T_1 is committed or aborted (as shown in the middle of Figure 4.2). We call this deferment *commit waiting*. Hereafter, we assume that

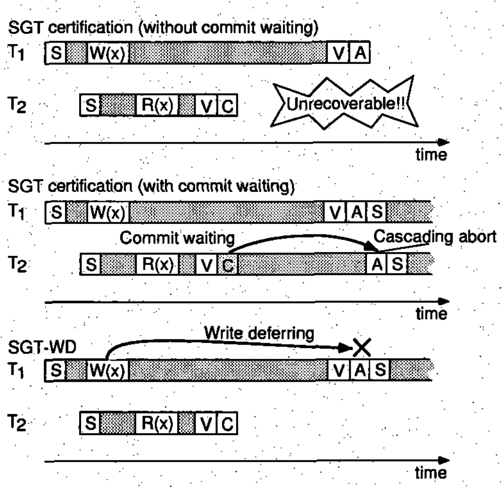

Figure 4.2: Comparison of SGT certification and SGT-WD (case 1)

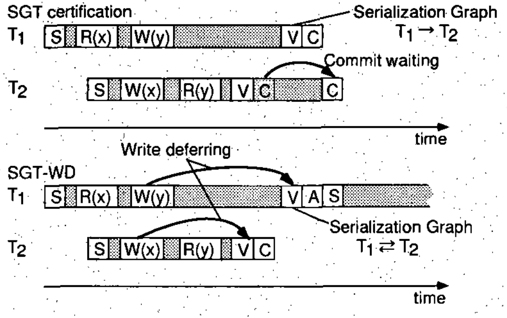

Figure 4.3: Comparison of SGT certification and SGT-WD (case 2)

SGT certification always carries out commit waiting if necessary. On the other hand, SGT-WD avoids such an execution by deferring the write operation of T_1. Under SGT-WD, x is not modified when T_2 reads it, while T_2 reads x modified by T_1 under SGT certification. Therefore, under SGT-WD, T_2 does not need to be aborted when T_1 is aborted. Clearly, SGT-WD is more desirable than SGT certification in this case. Consider another execution, shown in Figure 4.3. In this case, we assume that there are no transactions except T_1 and T_2. Under SGT certification, T_1 and T_2 do not form a cycle. Under SGT-WD, however, a cycle is formed, because the real execution of W(y) of T_1 is deferred until the write phase. Therefore, T_1 should be aborted and restarted. In the case of SGT certification, such a cycle is not formed and neither T_1 nor T_2 is aborted. Therefore, it may be concluded that SGT certification has an advantage over SGT-WD in this case. However, there is commit waiting for SGT certification. That is, under SGT certification, the commit operation of T_2 must wait until T_1 is committed. This means that a transaction waits for another transaction.

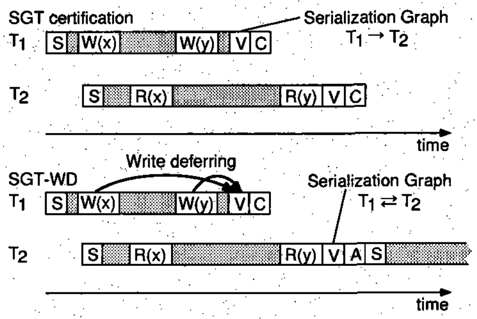

Figure 4.4: Comparison of SGT certification and SGT-WD (case 3)

Accordingly, the merit of certifications that aggressively schedule operations is damaged. On the other hand, no transactions wait for other transactions in SGT-WD. In this case, it is difficult to say which algorithm is more desirable.

The commit waiting mentioned above is also necessary for usual SGT. Moreover, most scheduling algorithms (including Two-Phase Locking and Timestamp Ordering) need this type of commit waiting to maintain the recoverability of executions[4]. Under SGT-WD, no commit waiting is needed, because all transactions read only data items written by committed transactions. This is one of the merits of SGT-WD.

In the case of Figure 4.4, SGT-WD is clearly worse than SGT certification. Under SGT certification, T_1 and T_2 do not form a cycle, and both are committed immediately (without commit waiting). Under SGT-WD, however, a cycle is formed, because the real execution of W(x) and W(y) of T_1 is deferred until the write phase. Therefore, T_2 should be aborted and restarted.

4.6.2 Overview of Simulations

SGT-WD is proposed to suppress the effect of scheduling cost to the performance of DBSs. We consider that SGT-WD has a effect especially for global transactions. In order to examine how useful SGT-WD is, the processing time of transactions under SGT, SGT certification, and SGT-WD is evaluated through simulations. The three types of scheduler is implemented on a simulator of a distributed database system. In distributed database systems, the communication cost strongly affects the processing time. To reduce the scheduling cost for local transactions, FT scheme described in Chapter 3 is used in all schedulers. The following assumptions are similar to Chapter 3.

1. A transaction is a string of operations.

2. There are 10 sites in the distributed database system.

3. 100 data items are stored at one site.

4. The transaction size is fixed; that is, all transactions contain 8 read/write operations.

5. 25% of all operations are write operations.

6. Each global transaction accesses at most 3 sites.

Simulations are executed in steps. We measure the transaction processing time by the number of steps. We considered that the execution time of a transaction mainly consists of the computation time (including CPU processing and access to memory) used for scheduling, the communication delay for sending messages, and the I/O delay for data access. Among them, the computation time seems to be much smaller than the others. We assume that schedulers can process one message in one step, and that the access to a data item (including I/O delay) needs 100 steps.

To execute one operation, two types of messages are needed:

- Messages for scheduling, which are needed to search for local SGs
- Messages for data access, which are needed to transfer data values

The number of messages of the former type can be suppressed by the certification approach, while messages of the latter type are still necessary.

4.6.3 Simulation Results

In simulations, the following two factors are selected as parameters:

- The time needed to transfer an message between two sites (denoted *com-delay*).
- The mean interarrival time of transactions (denoted *arr-interval*).

Figure 4.5 depicts the mean processing time of global transactions when *arr-interval* is fixed to 100 steps. The processing time increases with the communication delay. The rate of increase of the usual SGT is higher than that of SGT-WD and SGT certification. Of course, the influence of the communication delay on the processing time depends on the number of communications. Since SGT-WD suppresses the number of communications by the certification approach, the influence of the communication delay is smaller in SGT-WD than

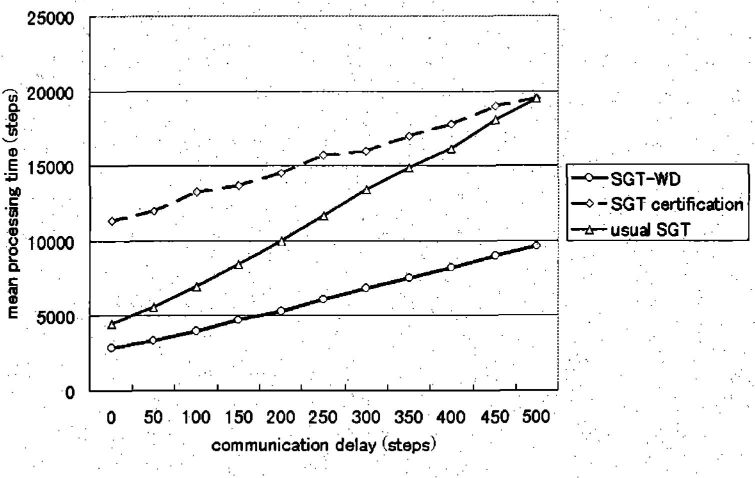

Figure 4.5: Mean processing time of global transactions. (arr-interval = 100 steps, locality = 80%)

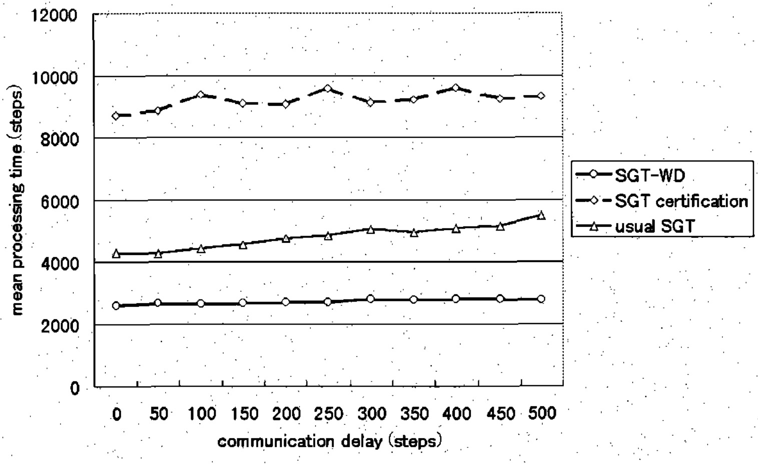

Figure 4.6: Mean processing time of local transactions. (arr-interval = 100 steps, locality = 80%)

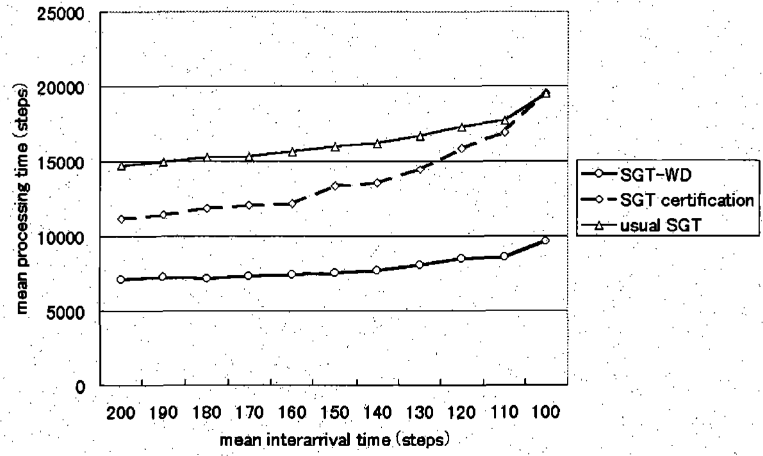

Figure 4.7: Mean processing time of global transactions. (com-delay = 500 steps, locality = 80%)

in usual SGT. Though the rates of increase of SGT-WD and SGT certification are almost equal, there is a large difference between the processing times of them. This is because SGT certification causes much more abortions than SGT-WD.

Figure 4.6 depicts the mean processing time of local transactions in the same case as Figure 4.5. This figure shows that processing time of local transactions is almost not affected by the communication delay. This means that the communications for local transactions is effectively suppressed by FT scheme. We consider that the difference of processing times of three algorithms is due to the difference of number of abortions.

Figure 4.7 depicts the mean processing time of global transactions when *com-delay* is fixed to 500 steps. In the figure, the mean processing time of SGT-WD is the shortest. The processing time of the usual SGT is long because *com-delay* is fixed to a large value. However, the increase rate of SGT certification is much higher than that of SGT-WD or the usual SGT. This is because SGT certification causes so many abortions under a heavy load. Figure 4.8 depicts the mean processing time of local transactions in the same case. For local transactions, the processing time of SGT certification is the longest. Since local transactions are hard to be affected by communication delay, the difference of the processing time is caused by the difference of the number of abortions. It is unexpected that the increase rates of SGT-WD is similar to that of the usual

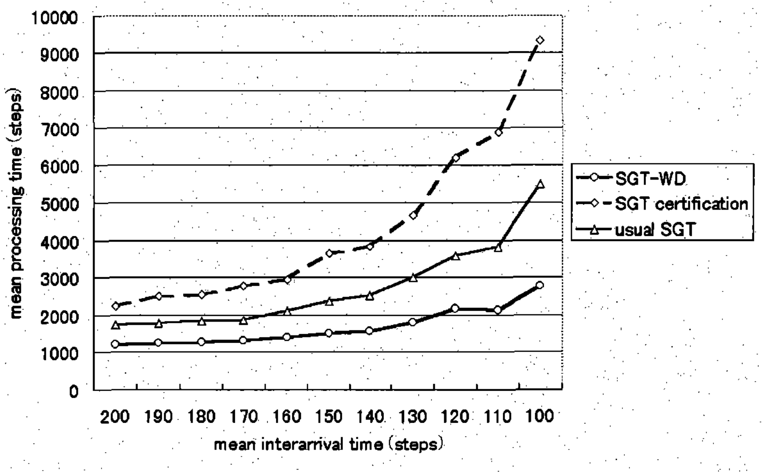

Figure 4.8: Mean processing time of local transactions. (com-delay = 500 steps, locality = 80%)

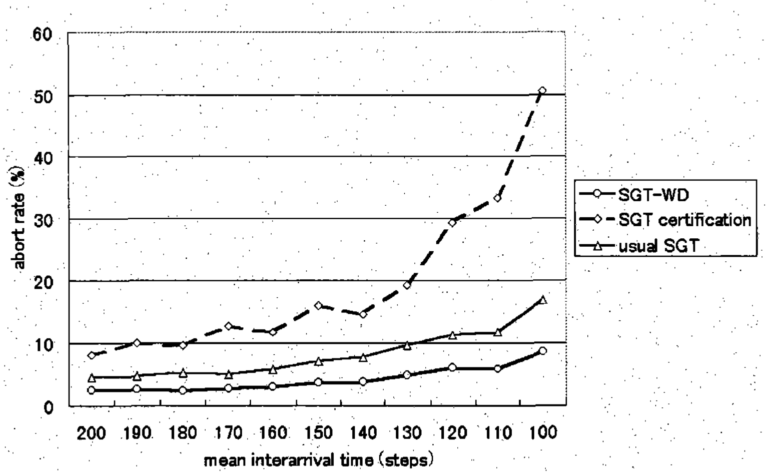

Figure 4.9: Abort rate. (com-delay = 500 steps, locality = 80%)

SGT. This means that though SGT-WD takes the certification approach, its tolerance to a heavy load is almost equal to that of the usual SGT. We consider that this is the benefit of cascadelessness which is brought by deferment of write operations. Figure 4.9 depicts the abort rate, where

$$\text{abort rate} = \frac{\#\text{ of abortions}}{\#\text{ of committed transactions}}$$

The abort rate of SGT-WD is lower than that of the usual SGT or SGT certification. Though SGT-WD adopts the certification approach, the number of abortions is suppressed, because any cascading aborts are avoided by deferment of writes. Note that the graphs in Figure 4.8 are similar to those in Figure 4.9. Since local transactions requires few communications, the number of abortions much affects the processing time of local transactions.

Simulation results showed that SGT-WD succeeded in obtaining tolerance of the communication delay of SGT certification without spoiling the load tolerance of the usual SGT.

4.7. Conclusions

In this chapter, a scheduling algorithm is proposed to suppress the effect of scheduling cost to system performance. We call our algorithm Serialization Graph Testing with Write Deferring (SGT-WD). SGT-WD is based on SGT certification to suppress the number of validations which involve the traverse of the SG. Dislike SGT certification, however, SGT-WD defers write operations by using internal buffers. Cascading aborts are avoided by this deferment of write operations. This is an advantage of SGT-WD over SGT certification. Though SGT certification can schedule any operations immediately, it often delays commitments of transactions waiting other commitments in order to avoid unrecoverable executions. This waiting time may cancel out the merit of immediate execution of operations in SGT certification. In SGT-WD, however, commit waiting is not required. Transactions can write into data items actually only when they are sure to be committed. That is, there exists no transactions which read from aborted transactions. Hence recoverability of concurrent execution is always preserved in SGT-WD. This is another advantage of SGT-WD.

On the other hand, as the result of deferring write operations, there exist cases that a transaction which was to be committed should be aborted. However, there also exist cases that a transaction which had to be aborted can be committed by write deferring. Therefore, it is not a disadvantage of SGT-WD.

We evaluated SGT-WD, the usual SGT, and SGT certification by means of simulations on distributed database systems. Two features of SGT-WD were recognized through these simulations. First, the influence of the communication delay on the processing time under SGT-WD is smaller than under the usual SGT, because of the suppression of communication by the certification approach. Second, SGT-WD is more tolerant of a load increase than SGT certification. This merit results from the deferment of write operations.

Chapter 5

Conclusions

5.1. Conclusions

In this thesis, we studied the distributed scheduling method which provides high concurrency without imposing special conditions on transactions. To provide high concurrency, we adopted SGT as the basis of our method. The major drawback of distributed SGT is large cost for scheduling which includes maintenance and traverse of the SG in each site of distributed DBSs. To overcome these drawback, we took two approaches. One is to improve the method for traverse of the SG so that the communication is suppressed. Another is to modify SGT itself to reduce the opportunity of scheduling.

In Chapter 3, the method for traverse of the SG, called *Fractional Tag Scheme* (FT), is proposed to suppress the communication cost of distributed SGT. Under FT, the SG is maintained in distributed manner, that is, each site maintains its local SG reflecting only the conflicts on the data items which are stored in the site. Therefore, update of local SG in each site can be done without any communication. In an SG traverse, a local SG is traversed at each site. Acyclicity of the global SG is determined by the results of local traverses which are gathered through message passing. To suppress the time of SG traverse, local SG traverses are performed simultaneously at several sites under FT. Fractional tags are introduced to detect the completion of whole traverse of the SG. Using fractional tags, we proposed the method for the distributed SG traverse. In Chapter 3, we show the correctness of FT and evaluate its performance by simulations in terms of the number of messages which are used for scheduling.

In Chapter 4, a variant of SGT is proposed to suppress the effect of the scheduling cost to the system performance. We have chosen SGT certification rather than usual SGT. SGT certification performs a validation, i.e. SG traverse, only once for each transaction, while each operation requires validation in usual

SGT. This feature of SGT certification is a great advantage in suppression of the effect of the scheduling cost to the performance. On the other hand, the delay of abortion in SGT certification may encourage cascading aborts. To eliminate cascading aborts, we integrated SGT certification with the idea of Optimistic Concurrency Control (OCC) which avoids cascading aborts by deferring substantial write operations using internal buffers. Thus we proposed a Serialization Graph Testing with Write Deferring (SGT-WD) which causes no cascading aborts. The feature of SGT-WD is (1) no cascading aborts occur, (2) the number of abortions is suppressed, and (3) data restoring is unnecessary when transaction is aborted for scheduling. In Chapter 4, the correctness proof and performance evaluation of SGT-WD are also presented.

It has been thought that SGT scheduling is so expensive that using SGT is unrealistic. However, appearance of new types of DBSs and surprising improvement of computers made SGT one possible choice. Nevertheless, in distributed DBSs, the communication cost is still a serious problem because traverse of the SG involves intersite communication. In this thesis, FT and SGT-WD are proposed to solve the problem. Simulation results showed that FT and SGT-WD are useful to suppress the scheduling cost of SGT in distributed DBSs. This means that the only drawback of SGT is relieved. Thus SGT is now a promising solution for concurrency control of recent and future DBSs, even if they are distributed.

5.2. Future Works

As new types of databases appear, a new model of transactions has also been proposed. As mentioned before, in new types of databases such as object oriented databases, there are transactions called long-lived transaction (LLT). If an LLT fails, its long lifetime causes large overhead for the rollback. The nested transaction is a new transaction model proposed to deal with such a problem. A nested transaction has a hierarchical structure. It consists of one top-level transaction and several subtransactions. Abortion and rollback can be done for each subtransaction. Most of scheduling algorithms for nested transactions are based on locking protocol and it seems useful to introduce SGT to improve concurrency. Therefore, we are now studying how to apply SGT for scheduling of nested transactions [30]. It is our future work to modify the method proposed in this thesis in order to apply it for scheduling of nested transactions.

References

[1] D. Agrawal, A. El Abbadi, and A.K. Singh. Consistency and orderability: Semantics-based correctness criteria for databases. *ACM Trans.Database Syst.*, 18(3):460–486, September 1993.

[2] P. Ammann, S. Jajodia, and I. Ray. Using formal methods to reason about semantics-based decomposition of transactions. In U. Dayal, P.M.D. Gray, and S. Nishio, editors, *Proc. of the 21th Intl. Conf. on Very Large Data Bases*, pages 218–227, Zurich, Swizerland, 1995.

[3] B.R. Badrinath and K. Ramamritham. Semantics-based concurrency control: Beyond commutativity. *ACM Trans.Database Syst.*, 17(1):163–199, March 1992.

[4] P.A. Bernstein, D.W. Shipman, and W.S. Wong. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, Reading, Massachusetts, 1987.

[5] C. Boksenbaum, M. Cart, J. Ferrié, and J. Pons. Concurrent certifications by intervals of timestamps in distributed database systems. *IEEE Trans. Software Eng.*, SE-13(4):409–419, April 1987.

[6] H. Boral and I. Gold. Towards a self-adapting centralized concurrency control. In B. Yormark, editor, *SIGMOD '84 Proceedings*, pages 18–32, Boston, MA, 1984. ACM, acm PRESS.

[7] M.J. Carey. Improving the performance of an optimistic concurrency control algorithm through timestamps and versions. *IEEE Trans. Software Eng.*, SE-13(6):746–751, June 1987.

[8] P. Dasgupta and Z.M. Kedem. The five color concurrency control protocol: Non-two-phase locking in general databases. *ACM Trans.Database Syst.*, 15(2):281–307, June 1990.

[9] M.H. Eich. Graph directed locking. *IEEE Trans. Software Eng.*, 14(2):133–140, February 1988.

[10] A.K. Elmagarmid, N. Soundararajan, and M.T. Liu. A distributed deadlock detection and resolution algorithm and its correctness proof. *IEEE Trans. Software Eng.*, 14(10):1443–1452, October 1988.

[11] A.A. Farrag and M.T. Özsu. Using semantic knowledge of transactions to increase concurrency. *ACM Trans.Database Syst.*, 14(4):503–525, December 1989.

[12] P.A. Franaszek, J.R. Haritsa, J.T. Robinson, and A. Thomasian. Distributed concurrency control based on limited wait-depth. *IEEE Trans. Parallel and Distributed Syst.*, 4(11):1246–1264, November 1993.

[13] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans.Database Syst.*, 8(2):186–213, June 1983.

[14] M. Herlihy. Apologizing versus asking perimssion: Optimistic concurrency control for abstract data types. *ACM Trans.Database Syst.*, 15(1):96–124, March 1990.

[15] G.S. Ho and C.V. Ramamoorthy. Protocols for deadlock detection in distributed database systems. *IEEE Trans. Software Eng.*, SE-8(6):554–557, November 1982.

[16] T. Ibaraki, T. Kameda, and N. Katoh. Cautious transaction schedulers for database concurrency control. *IEEE Trans. Software Eng.*, 14(7):997–1009, July 1988.

[17] A.D. Kshemkalyani and M. Singhal. Invariant-based verification of a distributed deadlock detection algorithm. *IEEE Trans. Software Eng.*, 17(8):789–799, August 1991.

[18] A.D. Kshemkalyani and M. Singhal. Efficient detection and resolution of generalized distributed deadlocks. *IEEE Trans. Software Eng.*, 20(1):43–54, January 1994.

[19] H.T. Kung and J.T. Robinson. On optimistic methods for concurrency control. *ACM Trans.Database Syst.*, 6(2):213–226, June 1981.

[20] P. Leu and B. Bhargava. Multidimensional timestamp protocol for concurrency control. *IEEE Trans. Software Eng.*, SE-13(12):1238–1253, December 1987.

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

[22] N. Natarajan. A distributed scheme for detecting communication deadlocks. *IEEE Trans. Software Eng.*, SE-12(4):531–537, October 1988.

[23] R. Obermarck. Distributed deadlock detection algorithm. *ACM Trans.Database Syst.*, 7(2):187–208, June 1992.

[24] M.T. Özsu and P. Valduriez. Distributed data management:unsolved problems and new issues. In T.L. Casavant and M. Singhal, editors, *Readings in Distributed Computing Systems*, pages 512–544. IEEE Computer Society Press, Los Alamitos, California, 1994.

[25] C.A. Papadimitriou, P.A. Bernstein, and J.B. Rothnie. Some computational problems related to database concurrency control. In *Proc. Conf. Theoretical Comp. Sci.*, pages 275–282, 1987.

[26] D.P. Reed. Implementing atomic actions on decentralized data. *ACM Trans. Computer Syst.*, 1(1):3–23, February 1983.

[27] K. Salem, H. Garcia-Molina, and J. Shands. Altruistic locking. *ACM Trans.Database Syst.*, 19(1):117–165, March 1994.

[28] M.K. Sinha, P.D. Nandikar, and S.L. Mehndiratta. Timestamp based certification schemes for transactions in distributed database systems. In S. Navathe, editor, *SIGMOD '85 Proceedings*, pages 402–411, Austin, Texas, 1985. ACM, acm PRESS.

[29] Y. Sugiyama, M. Fujii, T. Kasami, and J. Okui. Distributed deadlock detection algorithm. *Trans. IEICE*, 63-D(1):40–47, January 1980. (in Japanese).

[30] H. Tada, M. Higuchi, and M. Fujii. A model of nested transaction with fine granularity of concurrency control. In *Proc. 12th IEEE Pacific Rim Conf. on Communications, Computers and Signal Processing*, pages 977–980. IEEE, 1997.

[31] W.E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Trans. Comput.*, 37(12):1488–1505, December 1988.

[32] P.S. Yu, H. Heiss, and D.M. Dias. Modeling and analysis of a time-stamp history based certification protocol for concurrency control. *IEEE Trans. Knowledge and Data Eng.*, 3(4):525–537, December 1991.

List of Figures

2.1 A system configuration of a database system 10
2.2 A distributed database system. 10
2.3 Examples of transactions 14
2.4 An example of a history 14
2.5 An example of a serialization graph 16

3.1 An example of a global cycle. 28
3.2 An example of SG traverse. 32
3.3 An example of fractional tags. 33
3.4 An example of an useless message. 33
3.5 Average number of messages. 42
3.6 The distribution of numbers of messages (locality = 20%). . . . 43
3.7 The distribution of numbers of messages (locality = 80%). . . . 43

4.1 Comparison between OCC and SGT-WD 52
4.2 Comparison of SGT certification and SGT-WD (case 1) 53
4.3 Comparison of SGT certification and SGT-WD (case 2) 53
4.4 Comparison of SGT certification and SGT-WD (case 3) 54
4.5 Mean processing time of global transactions. (arr-interval = 100 steps, locality = 80%) 56
4.6 Mean processing time of local transactions. (arr-interval = 100 steps, locality = 80%) 56
4.7 Mean processing time of global transactions. (com-delay = 500 steps, locality = 80%) 57
4.8 Mean processing time of local transactions. (com-delay = 500 steps, locality = 80%) 58
4.9 Abort rate. (com-delay = 500 steps, locality = 80%) 58

List of Tables

2.1 The SGT algorithm 21

3.1 The distributed SGT algorithm 29
3.2 The distributed SGT algorithm (continued) 30
3.3 A function *SGT-validation*(T) using FT 34
3.4 The SGT-GC algorithm 38
3.5 The SGT-GC algorithm (continued) 39

4.1 The SGT-WD algorithm 50