INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Locking Granularity in Shared-Nothing Parallel Database Systems

by

Yan Gong, B.Sc.

A Thesis Submitted to
the Faculty of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science

Ottawa-Carleton Institute for Computer Science

School of Computer Science

Carleton University

Ottawa, Ontario

September 1998

© Copyright

1998, Yan Gong

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-36879-3

The undersigned hereby recommend to
The Faculty of Graduate Studies and Research
acceptance of the thesis,
Locking Granularity in Shared-Nothing Parallel Database Systems
submitted by
Yan Gong, B.Sc.
in partial fulfillment of the requirements
for the degree of Master of Computer Science

A handwritten signature in black ink, appearing to be 'E. Smith', written over a horizontal line.

Director, School of Computer Science

A handwritten signature in black ink, consisting of several loops and a long horizontal stroke, written over a horizontal line.

Thesis Supervisor

Carleton University
September 8, 1998

Abstract

Locking is the most common and effective concurrency control mechanism used to control concurrent accesses to a shared database in multiuser database systems. Locking granularity refers to the size of a lock, such as a record, a page, or a relation. An important issue that arises in the design of a database management system is the desired size of a lock. The choice of locking granularity presents a trade-off between the level of concurrency and the lock overhead. This trade-off affects the overall system performance.

This thesis studies desired locking granularity for shared-nothing parallel database systems using the simulation methodology. We simulate a shared-nothing database system on a network of workstations (NOW) to examine the performance of a NOW-based system as a parallel database system. Data contention and deadlocks, which were thought very low and rare in previous studies, are also studied.

Acknowledgments

I would like to take this opportunity to express the special gratitude to my supervisor, **Dr. Sivarama P. Dandamudi**, for his support and guidance, especially his patience and always being there to help and discuss with me.

I would like to thank my family, for their love and support to me.

Finally, I would like to thank my friends, **Raghunath Govindachari**, **Keith Cockwell**, **Cyrus Soleiman**, and **Jemal Abawajy**, for their precious discussions and suggestions.

Contents

Abstract	iii
Acknowledgments	iv
List Of Tables	viii
List Of Figures	ix
1 Introduction	1
1.1 Motivation	1
1.2 Concurrency Control	3
1.2.1 Problems without Concurrency Control	3
1.2.2 Concurrency Control Mechanisms	4
1.3 Locking Granularity and Concurrency	6
1.4 Contributions of the Thesis	8
1.5 Outline of the Thesis	9
2 Background	10
2.1 Parallel Transaction Processing	10
2.1.1 Properties of Transactions (ACID)	11
2.1.2 Parallelism in Transaction Processing	12
2.2 Parallel Database System Architectures	13

2.2.1	Shared-Everything Architecture	14
2.2.2	Shared-Disk Architecture	14
2.2.3	Shared-Nothing Architecture	16
2.3	Database Partitioning	16
2.4	Networks of Workstations (NOWs)	18
2.5	Two-phase Locking	20
2.6	Two-phase Commit Protocol	21
2.7	Deadlocks	23
2.8	Previous Work	27
3	Simulation Design and Experimental Environment	30
3.1	Database Model	30
3.2	Workload Model	32
3.3	System Model	33
3.4	Transaction Processing System Design	34
3.4.1	Overview	34
3.4.2	Transaction Generation	36
3.4.3	Transaction Management	38
3.4.4	Lock Management	41
3.4.5	Global Deadlock Detection	46
3.4.6	Resource Management	46
3.4.7	Network Management	46
3.5	Experimental Environment	47
3.5.1	NOW-based Parallel System	47
3.5.2	Parallel Virtual Machine (PVM)	48

4	Performance Evaluation	50
4.1	Simulation Parameter Settings	50
4.2	Validation of the Model	54
4.3	Performance with Distributed Lock Manager	55
4.3.1	Sensitivity to mpl	56
4.3.2	Sensitivity to Transaction Size	60
4.3.3	Sensitivity to System Size	65
4.3.4	Sensitivity to Think Time	70
4.3.5	Sensitivity to Data Skew	74
4.4	Performance with Centralized Lock Manager	79
4.4.1	Sensitivity to System Size	79
4.5	Summary	83
5	Conclusion	87
5.1	Summary and Conclusions	87
5.2	Future work	88
	References	90
A	Additional Data for Distributed Lock Manager (DLKM)	94

List of Tables

1.1	Lock Conflicting Table	6
4.1	Default Simulation Parameter Settings	52
4.2	Output Statistics	53

List of Figures

2.1	Parallel Database Architectures	15
2.2	Two-phase Locking	21
2.3	Two-phase Commit Protocol	23
2.4	Wait-For Graph	24
2.5	Global Deadlock WFG	25
3.1	System Model	35
3.2	Transaction Processing System Design	37
3.3	Data Structures of wait list, lock list and WFG	44
3.4	PVM Architectural Overview	49
4.1	Response time with DLKM (lock table in main memory)	56
4.2	Blocking ratio with DLKM (lock table in main memory)	57
4.3	Abort ratio with DLKM (lock table in main memory)	57
4.4	Lock overhead with DLKM (lock table in main memory)	58
4.5	Throughput with DLKM (lock table in main memory)	59
4.6	Response time with DLKM (lock table on disk)	60
4.7	Lock overhead with DLKM (lock table on disk)	61
4.8	Abort ratio with DLKM (lock table on disk)	62
4.9	Blocking ratio with DLKM (lock table on disk)	62
4.10	Response time with DLKM—small transaction (lock table in main memory)	64

4.11 Response time with DLKM—large transaction (lock table in main memory)	64
4.12 Response time with DLKM (lock table on disk)	66
4.13 Lock overhead with DLKM, (lock table on disk)	67
4.14 Abort ratio with DLKM (lock table on disk)	67
4.15 Blocking ratio with DLKM (lock table on disk)	68
4.16 Response time with DLKM (lock table in main memory)	69
4.17 Abort ratio with DLKM (lock table in main memory)	70
4.18 Response time with DLKM (lock table on disk)	71
4.19 Lock overhead with DLKM (lock table on disk)	71
4.20 Abort ratio with DLKM (lock table on disk)	72
4.21 Blocking ratio with DLKM (lock table on disk)	73
4.22 Response time with DLKM (lock table in main memory)	73
4.23 Lock overhead with DLKM (lock table in main memory)	74
4.24 Response time with DLKM (lock table on disk)	75
4.25 Lock overhead with DLKM (lock table on disk)	76
4.26 Blocking ratio with DLKM (lock table on disk)	76
4.27 Abort ratio with DLKM (lock table on disk)	77
4.28 Response time with DLKM (lock table in main memory)	78
4.29 Lock overhead with DLKM (lock table in main memory)	78
4.30 Response time with CLKM (lock table on disk)	80
4.31 Throughput with CLKM (lock table on disk)	80
4.32 Lock overhead with CLKM (lock table on disk)	81
4.33 Abort ratio with CLKM (lock table on disk)	81
4.34 Blocking ratio with CLKM (lock table on disk)	82
4.35 Response time with CLKM (lock table in main memory)	83
4.36 Throughput with CLKM (lock table in main memory)	84
4.37 Lock overhead with CLKM (lock table in main memory)	84

4.38	Abort ratio with CLKM (lock table in main memory)	85
4.39	Blocking ratio with CLKM (lock table in main memory)	85
5.1	A sample lock hierarchy	89
A.1	Throughput with DLKM, (lock table on disk)	95
A.2	Throughput with DLKM—small transaction, (lock table in main mem- ory)	95
A.3	Throughput with DLKM—large transaction, (lock table in main mem- ory)	96
A.4	Lock overhead with DLKM, (lock table in main memory)	96
A.5	Abort ratio with DLKM, (lock table in main memory)	97
A.6	Blocking ratio with DLKM, (lock table in main memory)	97
A.7	Throughput with DLKM, (lock table on disk)	98
A.8	Throughput with DLKM, (lock table in main memory)	98
A.9	Lock overhead with DLKM, (lock table in main memory)	99
A.10	Blocking ratio with DLKM, (lock table in main memory)	99
A.11	Throughput with DLKM, (lock table on disk)	100
A.12	Throughput with DLKM, (lock table in main memory)	100
A.13	Abort ratio with DLKM, (lock table in main memory)	101
A.14	Blocking ratio with DLKM, (lock table in main memory)	101
A.15	Throughput with DLKM, (lock table on disk)	102
A.16	Throughput with DLKM, (lock table in main memory)	102
A.17	Blocking ratio with DLKM, (lock table in main memory)	103
A.18	Abort ratio with DLKM, (lock table in main memory)	103

Chapter 1

Introduction

Relational databases have been playing a key role in the commercial information systems. A relational database is defined as a collection of logically interrelated data items that are organized as a set of formally described tables, also called relations. The data can be accessed or reassembled in many different ways.

1.1 Motivation

Recently, parallel database systems are receiving a greater amount of attention from both the academic research area and the commercial market. With the increasing amount of data to be stored and accessed, parallel database systems have already been recognized as the most effective solution to the problem of providing good response time to several users. Each user may access a large volume of data. In the traditional centralized database system, which logically has a single processor, accessing a large amount of data results in high response time and low system throughput.

A parallel database system consists of multiple processors that are connected by an interconnection network. These processors cooperate with each other to work on a single transaction in parallel. This greatly improves the performance of the sys-

tem. There are three types of parallel database systems: shared-everything, where all processors share a global memory and a collection of disks; shared-disk, where all processors share the disks but have their own private memory; and shared-nothing, where neither the memory nor the disks are shared among processors. This study is based on a shared-nothing system. Shared-nothing systems are more attractive mainly because of their scalability and reliability [31]. In a shared-nothing system, the database is partitioned and stored at different nodes physically. Database applications execute on multiple nodes. Each node processes a part of the application locally, independently and concurrently. Chapter 2 discusses the advantages and disadvantages of these three parallel database systems in detail.

A transaction is defined as a sequence of actions operated on the databases. Unlike queries, which only read data from the database, transactions can both read and write the data. In a shared-nothing database system, a transaction is split into several sub-transactions. Each sub-transaction executes at a node depending on where the data to be accessed resides. A transaction is not completed until all its sub-transactions complete.

In a multi-user database system, more than one transaction can access the shared database at the same time. Such transactions are called concurrent transactions. Concurrency control mechanism is responsible for managing the accesses of concurrent transactions to the shared database such that the consistency and correctness of transactions and database are preserved.

Concurrency control is an important aspect in a database management system (DBMS). Section 1.2 discusses the problems that arise without concurrency control and also describes the basic concurrency control mechanisms. This thesis addresses one aspect of concurrency control—**locking granularity**.

Before we describe what locking granularity is, we discuss the concept of locking first. Locking is the most popular and effective concurrency control mechanism. A database, which consists of a collection of data, is divided into several data units. Each data unit is assigned a lock. Before a transaction accesses the data, it has to require the associated lock. The DBMS controls the access of concurrent transactions

by granting or blocking the lock request.

A lockable data unit is also called a granule. Locking granularity refers to the size of a granule and therefore determines the number of locks in the database. A granule can be any size of a data unit such as a *record*, a *page* and a *relation*. The choice of locking granularity affects the system response time and throughput. Due to the significant impact of locking granularity on the performance of system, the impact of locking granularity on parallel database system is recognized as an interesting and challenging topic. This thesis explores the desired locking granularity in shared-nothing parallel database systems.

Parallel processing improves the system performance substantially, but existing parallel database systems are very expensive. The idea of using networks of workstations (NOWs) in place of parallel database machines is appealing due to this concern. A NOW is a set of heterogeneous computers connected by a network. A NOW minimizes the system cost by utilizing the already available hardware including processors, memory and disks. The NOW is a relatively new topic in the computing area. This thesis simulates a shared-nothing parallel database system on a NOW-based system. Chapter 2 describes the NOW in more detail and discusses the differences between parallel database systems and NOW-based systems.

1.2 Concurrency Control

Concurrency control is the activity of coordinating concurrent accesses to a shared database in a multi-user database management system to keep the consistency of transactions and the shared database.

1.2.1 Problems without Concurrency Control

Concurrency control is not necessary in sequential database management systems where only one transaction can access the database at one time. In multi-user database systems, the database and transactions will be consistent without any concurrency control mechanism if:

- The database is read only. No update can be performed to the shared database.
- Concurrent transactions always access different data in the database. Inconsistency occurs only when concurrent transactions try to access the shared data in a conflicting mode.

Otherwise, when more than one transaction executes on shared data concurrently, they may cause the following inconsistency problems:

1. **Lost Updates:** When more than one transaction updates the shared data, undoing one transaction will also undo the update of others that have not been committed.
2. **Dirty Read:** If one transaction T1 reads a shared data that was previously updated by another transaction T2, then if T2 aborts, the data that T1 has read becomes non-existing data.
3. **Non-repeated Read:** One transaction reads the shared data, then another transaction updates the shared data. If the first transaction reads the shared data again, it will get a different value for the same data.

The first problem occurs when two or more transactions write the database concurrently. Allowing read and write of concurrent transactions to the shared data at the same time will lead to the second and third problems. The concurrent accesses of transactions should be somehow regulated.

The key point of concurrency control is to keep the *serializability* of concurrent transactions. Concurrent transactions are serializable if the outcome of a collection of concurrent transactions is equivalent to the one produced by running the same transactions sequentially.

1.2.2 Concurrency Control Mechanisms

Many concurrency control mechanisms have been designed and developed. In [4], Bernstein and Goodman survey 48 concurrency control methods. Most concurrency

control methods fall into one of the three mechanisms: locking, timestamp ordering, and optimistic.

- **Locking:** Locking realizes the access control by associating a lock to a unit of data. When a transaction wants to access the data, it has to acquire the lock for that data first. A transaction is blocked if its request conflicts with other transactions that hold the lock.
- **Timestamp Ordering:** Each transaction is assigned a unique timestamp. The timestamp of a transaction is usually determined by the arrival time of the transaction to the system. All operations use the timestamp of their transactions. Conflicting operations are executed in the timestamp order.
- **Optimistic:** It is also called *certificate*, which allows all transactions to read and write freely without being blocked till the commit stage. A transaction is committed only if it does not have any conflict with any of the transactions that have committed since it started execution. Otherwise, the transaction is aborted and restarted.

Locking is the most common and popular technique used to achieve concurrency control. Several commercial and experimental systems, such as the Non-Stop SQL from Tandem [32], the Gamma machine from the University of Wisconsin [11] and the Bubba at MCC [19], use lock-based concurrency control mechanisms.

The approach of physical lock is used in this thesis. Typically all locks in the database have the same size. There are two lock modes: read and write. Before reading a piece of data, a transaction has to get and set a read lock. Similarly, before writing data, a transaction has to get and set a write lock. If the requested lock modes of two transactions to the same lock are compatible, they can be granted concurrently. Otherwise one of them has to wait till the other finishes and releases the lock. Table 1.1 shows that lock conflict occurs only when there is a write lock request.

A transaction may request more than one lock. All the locks needed by the transaction can be requested either statically or dynamically.

	READ	WRITE
READ	no conflict	conflict
WRITE	conflict	conflict

Table 1.1: Lock Conflicting Table

- **Static Locking:**

All needed locks are requested at the beginning. A transaction can not be processed unless it has all the required locks. Once a transaction is activated, it can execute to completion without blocking. Deadlock occurs when two or more transactions wait for locks that are held by each other. Obviously, deadlock is impossible in static locking. The disadvantage of static locking is that it may cause starvation when a transaction can never get all the locks it needs. When a lock request fails, all locks must be requested again. This generates additional cost.

- **Dynamic Locking:**

In contrast to static locking, locks are requested on demand during execution. A transaction may be blocked after activation when the required lock is held by another concurrent transaction. Deadlock may also occur when several transactions wait for each other. The typical solution to deadlock is to abort and restart one or more transactions in the deadlock. Dynamic locking avoids the source of overhead in static locking, but must substitute the overhead associated with the work lost by restarting the deadlocked transactions.

1.3 Locking Granularity and Concurrency

This section discusses the main issue studied in the thesis. As mentioned before, a granule is a lockable unit. A granule can be a *record* as in System R [3] and DMS-1100 [14], a *page* as in IMS [14], a *column of a relation* as in INGRES [30], a *relation*,

or the whole *database* as in SYSTEM 2000 [29]. In all cases, data processing can not proceed if a required granule is locked by some other transactions in a conflict mode. *Record* is the finest granularity while *database* is the coarsest granularity. Both coarse granularity and fine granularity have their benefits and liabilities.

Since coarse granularity covers much data, a transaction does not need to set many locks. Thus the overhead of setting and releasing locks is low. However, with coarse granularity, a transaction may lock more data than it needs and cause unnecessary blocking of other transactions. For example, at file level granularity, a transaction will lock the whole file even if it accesses only few records in the file. Then other transactions that want to access other records in the file can not proceed. Coarse granularity reduces the number of transactions that can run concurrently and may cause the poor system performance. When the locking granularity is at the database level, transactions access database sequentially.

With fine granularity, the degree of concurrency is increased since more locks are available and can be requested by more concurrent transactions. On the other hand, fine granularity also increases the lock and unlock operations, thus more lock overhead. When a transaction accesses a large amount of data, the high lock overhead resulting from fine granularity may outweigh the advantage of high concurrency and lead to poor system performance.

Another issue related to locking granularity is the storage space for the lock table. The lock table has an entry for each lock. With coarse granularity, a small number of locks can be maintained in the main memory. With fine granularity, such as a record, the lock table with a large number of locks may have to be kept on the second storage device. In this case, the lock I/O overhead will affect the system performance.

There is a trade-off between the degree of concurrency and lock overhead depending on the granularity. Fine granularity has higher degree of concurrency but also high lock overhead. Coarse granularity minimizes the lock overhead by sacrificing the degree of concurrency. This trade-off determines the system performance and is decided by the choice of locking granularity.

1.4 Contributions of the Thesis

Locking granularity has been studied in centralized database systems with static locking [25] and dynamic locking [26] and in parallel database systems with static locking [9]. The impact of locking granularity on parallel database system with dynamic locking has not been studied. This gap is filled by this thesis.

This thesis adopts the approach of simulation. An extended model from the model in [25] is presented in this study. Compared with other studies, the model and assumptions in this thesis are more realistic. The main differences between this thesis and the previous studies are as follows:

- The database in the previous papers is modeled as a collection of data entities. The database in this thesis is more practical and modeled as a collection of relations with each relation having a number of records.
- Based on the model of database, the granularity in this thesis can be a *record*, a *page*, or a *relation*. In the previous studies, the granularity is a group of data entities.
- Other studies assume that any lock request for the same lock is exclusive. This thesis has read and write locks, and two read lock requests can be granted concurrently.
- In other studies, lock conflict is modeled analytically. The lock overhead including lock I/O and CPU overhead is set as an input parameter. In this thesis, locks are granted by really going through the lock table. Lock conflict is determined at run time. Lock conflict occurs when two transactions request one lock in a conflicting mode. The lock manager handles the requests. Lock CPU overhead is measured instead of being assumed.

Furthermore, all the other studies assume that the data contention is very low. In [26], even though dynamic locking is applied, deadlock is not studied at all.

This thesis creates an environment with data contention and examines the impact of deadlocks.

Lastly, the low cost of NOW-based system makes itself more attractive. As a new research area, NOW-based systems have not received much attention in the database community. This thesis investigates the performance of a NOW as a parallel database system.

1.5 Outline of the Thesis

Chapter 2 introduces the background related to this thesis including parallel database architectures, networks of workstations, two-phase locking and two-phase commit protocols. Previous work in this area is also discussed. Chapter 3 describes the database model, workload model and system model. The system design including the structure and behavior of the main components in the system, and the algorithms and policies employed are presented in this chapter. The chapter also describes the experimental environment. Chapter 4 presents and discusses the results of the experiments. Chapter 5 summarizes this thesis and gives the future work in this area.

Chapter 2

Background

Parallel database systems are mainly classified into three architectures: shared-everything, shared-disk and shared-nothing. Several partition schemes can be applied to distribute the database to several nodes in a parallel database system. There is a trend to use networks of workstations (NOWs) as a parallel database system, but there are some significant differences between a NOW-based system and a parallel database system. Two-phase locking and two-phase commit are considered as the locking scheme and commit protocol to ensure the consistency of transactions and the database. Deadlock is a typical problem with two-phase locking and has significant impact on system performance. This chapter describes the above issues in detail. The previous studies on locking granularity are also reviewed.

2.1 Parallel Transaction Processing

Transaction processing (TP) applications have been applied in several areas, such as business, government, communication, etc. When a person withdraws money from a banking machine, sells a book, or makes a phone call, a transaction is processed.

The idea of processing a transaction in parallel is appealing for several reasons. the main reasons are the increasing demand for high-speed transaction processing, the requirement for multiple accesses to the databases, and the large amount of data involved in processing these transactions. Furthermore, transactions are ideally

suited to parallel execution since usually a transaction consists of a collection of independent operations.

2.1.1 Properties of Transactions (ACID)

To understand transaction processing well, we need to know the four critical properties of transactions (ACID) [17]:

- **Atomicity:** A transaction executes either completely or not at all. In other words, a transaction may only be either committed or aborted. All changes of the transaction to the database will become durable and unchangeable if it is committed. Otherwise the database will return to the original state without any modification.
- **Consistency:** The work between the begin and commit of a transaction must be performed correctly. It ensures that the database with consistent states will still be consistent after executing the transaction.
- **Isolation:** The technical definition of isolation is *serializability*. A set of transactions is isolated if the effect of running them concurrently is the same as sequentially.
- **Durability:** Once a transaction is committed successfully, all the changes that it makes to the database becomes public and permanent. These changes can survive a failure of the transaction processing system, such as a system crash.

When designing a TP system that processes and manages transactions, the above four properties have to be ensured to maintain the correctness and consistency of transactions and the database. Isolation is the key property and criterion to be considered in concurrency control mechanism as discussed in [15]. The purpose of concurrency control is to maintain the serializability of concurrent transactions.

2.1.2 Parallelism in Transaction Processing

Transactions can be executed in parallel with two kinds of parallelism. One is inter-transaction parallelism, and the other is intra-transaction parallelism [12] [8].

- **Inter-transaction Parallelism**

Inter-transaction parallelism represents the parallel relationship between transactions. Inter-transaction parallelism is characterized by allowing many transactions to execute simultaneously. It enables multiple users to access the resources of the database concurrently.

Since neither the number nor the speed of processor changes, the execution time of a transaction will still be the same as without applying any parallelism. The response time of a transaction may be decreased by utilizing the idle time of processors and reducing the associated waiting time.

- **Intra-transaction Parallelism**

One way to reduce the execution time is to apply multiprocessors to execute a transaction in parallel. Intra-transaction parallelism exploits parallelism within a transaction. With intra-transaction parallelism, a transaction is partitioned into several sub-transactions that are distributed and executed in several processors independently. Both the response time and execution time of a transaction are reduced.

Intra-transaction parallelism is implemented on parallel database systems. There are two ways to implement the intra-transaction parallelism in a transaction processing system.

- *Program parallelism*: executes multiple operations of a given transaction in parallel.
- *Data parallelism*: operates a single operation concurrently and independently on different pieces of data that usually reside in different nodes.

Intra-transaction parallelism is suitable for transactions since usually transactions consist of many small independent operations. This feature of transactions suggests that a transaction can be easily divided into several sub-transactions and executed on different data in parallel. Besides, the more powerful high-speed network also facilitates the applying of intra-transaction parallelism.

In this thesis, both inter-transaction and intra-transaction parallelism are applied to improve the system performance.

2.2 Parallel Database System Architectures

Parallel database system may improve the system performance substantially. Much attention has been paid to the design of parallel database systems. A lot of research has been done in this area. Several database machines in the form of commercial products and research prototypes are available, such as Non-Stop in Tandem [32] and Gamma at the university of Wisconsin [11].

Parallel database systems combine the concepts of parallel processing and database management system. A parallel database system can be built by connecting together many processors, memory units, and disk devices through a global interconnection network. Several papers [12] [9] [8] discussed the design of parallel database systems.

General parallel processing systems can be divided into either shared-memory or distributed-memory systems. Shared-memory systems, also called multiprocessors, provide a global single address space. Processors communicate through shared memory variables. In distributed-memory systems, also called multicomputers, each processor has its own local memory. Processors communicate by sending messages.

Depending on the interconnection of components, parallel database systems are classified into three architectures: shared-everything (SE), shared-disk (SD) which are like the shared-memory system and share-nothing (SN) which is more like the distributed-memory system. Figure 2.1 shows the architectures of these three parallel database systems. Each architecture has its advantages and disadvantages. Rahm studied the structures of shared-disk and shared-nothing in [24]. Stonebraker

compared the three architectures in several aspects such as the difficulty of concurrency control, the difficulty of load balancing, the ability to scale to large systems, etc in [31]. Generally, the desired architecture in the design of a parallel database system depends on different applications.

2.2.1 Shared-Everything Architecture

In a shared-everything (SE) system, multiple processors share a global memory address space and all disks. In other words, the memory and disks are accessible directly by all processors. There are several parallel processing systems that are based on shared-memory architecture, such as IBM/370, Digital VAX, and Sequent Symmetry.

This architecture provides high-speed data access. Since all processors access the same memory space, it may cause interference among different processors and increase of the access latency. SE is also limited by its scalability. Adding more processors to the system induces more load on the interconnection work.

2.2.2 Shared-Disk Architecture

Compared with shared-memory, the shared-disk (SD) architecture is constructed by multiple processors sharing a common collection of disks but each has its private memory instead of sharing the memory. SD is the approach used in IBM's IMS/VS Data Sharing product, DEC's VAX database management system.

SD architecture is not very effective for database applications that read and write a shared database. When a process accesses the shared data, it gets a copy of it. To keep the consistency of other copies of the shared data, the process that updates the data has to inform other processes that have a copy of the data. This involves large amount of exchanging of data and messages. This also creates heavy traffic on the shared interconnection network. As with the SE architecture, SD architecture also does not scale well for database applications because the interconnection network

Figure 2.1: Parallel Database Architectures

must have the bandwidth that is the sum of that of the processors and disks. It is difficult to build such network that can scale to thousands of nodes.

2.2.3 Shared-Nothing Architecture

As its name, shared-nothing (SN) database systems share neither a global memory nor disks. Each disk is connected to a single processor and each processor has its own private memory. Resources are partitioned and distributed to all processors and can only be accessed locally. Processors communicate with each other by sending messages explicitly via the interconnection network. The SN architecture was pioneered by Teradata in the late 1970s. This design is used by Teradata, Tandem, NCR and Oracle/nCUBE. Commercial machines such as Non-Stop SQL from Tandem uses inter-transaction parallelism and DBC/1012 from Teradata uses intra-transaction and inter-transaction parallelism. Gamma machine that is used for research and built in University of Wisconsin also employs both of the transaction parallelisms.

SN architecture is attractive mainly because of its scalability and reliability. Since there is no resource sharing, the problem of interference that exists in SE systems is minimized in SN systems. The disadvantages of SN are the cost of communication and the data skew problem. Data skew is a typical problem with the SN database system. This problem is addressed in [31]. There are two ways to cause this problem. *Static skew* is observed when data are stored non-uniformly across a subset of disks. *Dynamic skew* is observed due to non-uniform data accesses across different processors by the transactions. Both types of uneven distribution can lead to some processors becoming potential bottlenecks. This thesis also studies the data skew problem from the perspective of *dynamic skew* within the context of lock management.

2.3 Database Partitioning

A database is a collection of relations, also called tables. Each row in the table represents a collection of related data values. All data in the same column have

the same attribute. A tuple, or record, is a row of values in the table. Partitioning a database involves distributing the tuples of each relation over several disks. To effectively use parallel database systems, the database should be partitioned across the disks in such way that the I/O bandwidths of multiple disks are utilized more efficiently. A database is partitioned into several fragments that are stored in disks at different nodes. Relations can be fragmented either *horizontally* based on the value of each row or *vertically* based on the value of each attribute. Typically, all relations are horizontally partitioned across all disk units in a parallel database system. The main partitioning strategies are round-robin, hash partitioning, range partitioning and random partitioning [12].

- **Round-robin:** This strategy distributes the tuples evenly in a round-robin fashion among all disk drives. Teradata and Gamma machines apply this strategy to partition the database. The round-robin strategy works well if all applications access a relation by sequentially scanning all of it (sequential access). A problem occurs when applications frequently access the tuples with some particular attribute values (associative access). When accessing a disk, there is a starting overhead associated with each access at each disk. With associative access, the round-robin strategy maximizes the starting overhead and may affect the system performance.
- **Hash partitioning:** Tuples are placed by applying a hashing function mapping on an attribute of each tuple. It is suited to applications that access the relation either sequentially or associatively. Associative access to tuples with a specific attribute value can be directed to only one disk and avoid the starting overhead on multiple disks. Parallel processing is achieved by processing different associative accesses with different distribute values in parallel.
- **Range partitioning:** Tuples with similar attributes, that are usually accessed frequently together, are placed together in the same partition. Tuples are clustered and distributed by a specified range of key values. Bubba, Gamma, and oracle provide range partitioning. Range partitioning is suited to

both sequential and associative access, but may cause the data skew problem when most data is distributed to one disk. Range partitioning may also cause the *execution skew* problem when an application only wants to access the data with the same attribute.

In this thesis, the round-robin strategy is chosen as the database partitioning strategy. Due to the fact that a transaction usually consists of a sequence of independent requests, the hash partitioning is not advantageous in this study. The problem of data skew makes the range partitioning undesirable. The data skew problem of shared-nothing system is studied by varying the distribution of data accessed by each transaction while keeping the partitions of the database evenly.

2.4 Networks of Workstations (NOWs)

The main concerns about a computing system are the high performance, sustained productivity, and low cost. A MPP combines a few hundred to a few thousand CPUs connected to hundreds of gigabytes of memory. MPPs offer enormous computational power, but the cost is a main disadvantage of MPPs. Large MPPs typically cost more than \$10 million. From another point of view, a large fraction of processing power available on a network of workstations (NOW) is wasted by idling the workstations. By using these workstations, we can build a virtual parallel machine that utilizes the wasted cycles in workstations to execute parallel jobs. This will meet the requirements of high-performance and low cost at the same time. Advanced technologies that provide inexpensive, low latency, high bandwidth, and scalable interconnection networks also facilitate the use of NOWs. NOWs not only provide additional processor cycles to be used, but also add substantial main memory for processing. Heterogeneity and time-sharing are two characteristics that distinguish NOW-based systems from conventional multiprocessor/multicomputer systems, that are homogeneous and dedicated. PVM and MPI are the available public domain software to configure a parallel virtual machine over a NOW.

Commercial parallel databases are also very expensive. Using a NOW-based system as a parallel database system and executing transactions in parallel can solve this problem. There are several differences between NOW-based systems and parallel database systems [8].

1. There is no central control in NOW-based systems. For example, the user can reboot the workstations. Therefore the database should not be distributed and stored across workstations. However, storing the database in one node, as in centralized database systems, will cause overhead because the data has to be sent to the workstations whenever a transaction wants to access the database. Parallel database systems do not have this problem since there is a central control and relations can be stored at each node locally.
2. NOW-based systems are heterogeneous. Workstations in a network can be of different types, such as PCs and powerful workstations. In a parallel database system, all nodes are typically homogeneous.
3. In NOW-based systems, more than one user can login to a workstation and run their own jobs. Thus, the load on each workstation will be different and has no central control. In parallel database systems, the load on each node can be controlled by a central scheduler, thus is more stable than NOW-based systems. In a dedicated NOW system, there is no background load.
4. NOW-based systems use the general purpose LANs as the interconnection network, while parallel database systems have a dedicated, special purpose fast communication network. Therefore, NOW-based systems will involve more communication overhead than parallel database systems due to the different network bandwidths.

This thesis assumes that the database is partitioned to individual workstations. The dedicated NOW system is used to mimic a SN parallel database system for database processing.

2.5 Two-phase Locking

Two-phase locking (2PL) is the most widely used concurrency control algorithm in the existing database systems, especially in shared-nothing database systems. 2PL is a dynamic locking scheme where a transaction requests a lock on a data when it needs the data. In [7], the performance issues of concurrency control mechanisms are studied. The 2PL provides the best performance in shared-nothing systems, followed by the basic timestamp ordering and the optimistic algorithm. Other studies [4] [27] also discuss the performance and implementation issues about 2PL. This thesis considers the 2PL as the concurrency control scheme.

In centralized database systems, concurrent accesses to the shared database are controlled by a concurrency control mechanism that enforces the *serializability* of the concurrent transactions. In parallel database systems, since a transaction executes on multiple nodes independently, a global *serializability* should be maintained by the concurrency control mechanism. A group of parallel transactions are serializable if and only if:

- The execution of the group of transactions is serializable at each node.
- The serialization order of these transactions is identical at each node.

2PL is designed to meet this requirement. There are two phases in the 2PL scheme:

- **Growing phase:** During this phase, a transaction requests locks without releasing any lock.
- **Shrinking phase:** A transaction may release locks, but may not obtain any new locks.

Figure 2.2 shows the two phases of the 2PL scheme. In the growing phase, the number of locks held by the transaction increases. At a certain point, the transaction acquires all locks and finishes processing. It then enters the shrinking phase

and prepares to commit. A transaction will release all locks when it terminates. Note that no sub-transaction can release any lock if other sub-transactions are still requesting lock. 2PL handles the parallel transactions very well, but suffers from the problem of deadlock. An effective method for detecting deadlock and an appropriate standard for choosing the victim are very important when using 2PL.

Figure 2.2: Two-phase Locking

2.6 Two-phase Commit Protocol

Two-phase commit (2PC) protocol is designed for parallel database systems to coordinate sub-transactions at different nodes to either commit or abort. The problem with parallel database systems is that one sub-transaction can commit at one node, but another sub-transaction in another node may fail before it commits. The reasons of failure are usually because of a local deadlock or system crash. In the case of deadlock, the transaction may be aborted and all of the changes made by the aborted transaction must be undone. In the case of system crash, after the system

recovers, the transaction should be able to commit. With 2PC, the global transactions can be terminated consistently. 2PC is the simplest atomic commitment protocol and is the most widely used in practice [10] [20].

Typically, there is a central coordinator to perform this protocol. When running 2PC, the coordinator sends out two rounds of messages. Each round is for each phase of the commitment activity as shown in Figure 2.3. In the first phase, the transaction coordinator exchanges one round of messages with the participants to collect the unanimous votes for commit or abort. Then it prepares to commit by writing the results of the transaction to disk, but not actually commit the transaction. In the second phase, the coordinator exchanges another round of messages with all participants to commit or abort the transaction as voted. No change is made to the state of the data at a participant node until the second phase. Since all changes are written to the durable second storage space, the consistency of the database can still be maintained under the appearance of system failures. 2PC is required whenever a transaction executes at more than one node. The 2PC adds more communication overhead, but the parallel processing brings more benefit. The basic 2PC protocol is given in the following:

1. The first phase starts when the coordinator sends a `COMMIT_REQUEST` message to each participant of the transaction to be committed.
2. The coordinator waits for all participants to “vote” on the request.
3. In response to receiving a `COMMIT_REQUEST` message, each participant votes by sending back the coordinator a message.
 - (a) `VOTE_COMMIT`, if it is prepared to commit.
 - (b) `VOTE_ABORT`, if it can’t prepare to commit for any reason.
4. The second phase begins when the coordinator receives the response from participants. If the coordinator receives `VOTE_COMMIT` from all participants, it will send a `COMMIT` message to all participants. The transaction is now officially committed. If the coordinator receives a `VOTE_ABORT` message

from any of the participants, it will send ABORT message to all participants, and the transaction is aborted.

5. Each participant will either commit or abort depending on the message received, then acknowledge the coordinator.

Figure 2.3: Two-phase Commit Protocol

2.7 Deadlocks

Deadlock is a typical problem with dynamic locking. Since this thesis employs the two-phase locking as the concurrency control method, the deadlock has to be handled. Deadlock occurs when more than one transaction waits for each other. In this case, none of them can ever proceed and some of them have to be restarted. The overhead cost of restarting the deadlocked transactions may affect the system performance significantly.

Several papers [15] [27] describe the formation and solution of deadlocks. Deadlocks are usually characterized in terms of a wait-for-graph (WFG). A WFG is a

directed graph where each vertex represents a transaction and each link represents the waiting relationship between the two vertex transactions connected by it. The source transaction is the waiting transaction. The end transaction is the transaction the source is waiting for. There exists a deadlock if and only if there is a cycle in the WFG. As shown in Figure 2.4, transaction t1 is waiting for t2, t2 is waiting for t3 and t4, and t3 are waiting for t1. A cycle exists among t1, t2, and t3. No transaction in this cycle can proceed.

Figure 2.4: Wait-For Graph

A shared-nothing database system involves multiple processors. Deadlock may occur either within a node, called local deadlock, or across several nodes, called global deadlock. Global deadlock may occur even though there is no local deadlock.

Figure 2.5 shows a simple example of WFG of a global deadlock. There are two transactions, t1 and t2, in a system with two nodes. t11 denotes the sub-transaction of t1 in node1, and the same denotation for t12, t21, t22. There is no cycle within any node, but there is a deadlock across the two nodes. The dot arrow lines indicate that t12/t21 is waiting for t11/t22 to complete. The waiting sub-transactions are not waiting for any lock, instead they are waiting for the completion of the other sub-transactions. In shared-nothing system, a transaction can complete and release the locks only when all its sub-transactions complete. This waiting may because of long execution time at one node. The different computation speed and load, which are typical in NOW-based systems, and the different sub-transaction size at different nodes affect the execution time of sub-transactions.

Figure 2.5: Global Deadlock WFG

There are three strategies to handle the deadlock: timeout, deadlock prevention and deadlock detection.

- **Timeout:**

One of the simplest ways to resolve deadlock is to specify a maximum wait time. A transaction is restarted if it waits beyond this time before its request is granted.

- **Deadlock prevention:**

Deadlock is prevented by never allowing blocked states that can lead to a circular waiting. Typically each transaction is associated with a priority. When conflicting occurs, the transaction with higher priority can wait. Otherwise it has to be restarted. This approach does not allow cycle because transactions with lower priority would never wait for higher priority transactions. The disadvantage of this approach is that it may abort transactions that are not really deadlocked. The Wound-wait, Wait-die, Immediate-restart and Running-Priority are deadlock prevention schemes.

- **Deadlock detection:**

Deadlock detection allows deadlock to occur temporarily. Deadlocks are detected by explicitly constructing the WFG and searching it for cycles. When there is a deadlock, a victim will be chosen to restart. There are several standards for choosing the victim.

1. *current blocker*: The transaction which is blocked most recently.
2. *random blocker*: Picked randomly among the transactions in the deadlock cycle.
3. *min lock*: The victim is holding the fewest locks among the deadlocked transactions.
4. *youngest*: the transaction with the most recent initial startup time.
5. *min work*: The transaction which has consumed the least amount of physical resources.

Timeout and deadlock prevention can easily restart a non-deadlocked transaction and thus affect the system performance. Furthermore, in practice, deadlocks have been observed rare [25]. This thesis chooses deadlock detection as the solution.

Local deadlock is detected whenever a request is blocked. Global deadlock detection is performed periodically. A central deadlock detector is responsible for detecting the global deadlock. The deadlock detector periodically collects the local WFGs and combines them into one global WFG. In a high data contention environment, transactions tend to compete for the resources at all time. In this case, continuous deadlock detection outperforms periodically detection. This is because that the longer the detection period is, the longer the locks are held by a deadlock transaction, and the more excessive blocking occurs. This thesis adopts the method of periodically deadlock detection based on two reasons. First, the deadlock detector collects local WFGs by sending and receiving messages, thus the cost of messages makes periodic detection cheaper. Second, global deadlock is considered rare.

Among the victim selection criteria, the minimum number of locks is found better than the other standards. This is because, if a transaction that holds a large number

of locks is restarted, it has to contend for all of its locks again and increases the probability of blocking and deadlock.

2.8 Previous Work

Locking granularity has been studied in both centralized and parallel database systems with either static locking or dynamic locking [25] [26] [9]. Among all the studies, the number of locks requested by a transaction is decided by the number of entities to be accessed. The mapping from number of entities to number of locks is called *lock placement*. There are three lock placement strategies.

- *Well lock placement*: When a transaction requests locks, the number of locks is decided in such way that the minimum number of granules that could possibly lock the number of required entities are requested. For example, if the granularity size is 5 entities, one transaction that wants to access 5 entities will only request 1 lock. This assumption is reasonable for transactions which access the database sequentially.
- *Worst lock placement*: It is the opposite of *well lock placement*. Each transaction requires the maximum number of locks possible. Applying the above example, the transaction will request 5 locks instead of 1 lock. This is to simulate transactions whose access pattern is the worst possible one from the point of view of locking mechanism.
- *Random lock placement*: Random lock placement is between the well and worst lock placement. A transaction requests locks randomly. In the above example, the number of locks to be requested will be between 1 and 5.

Static locking in centralized database:

This topic was studied by Ries and Stonebraker in [25]. A simulation model for centralized database management system is created in this study. There is only

one processing node where the whole database resides. The database is modeled as a collection of data entities. Locking granularity is varied from one to the total number of data entities. There is only one lock mode allowed in the database management system, *exclusive* mode. Any concurrent requests to the same lock can not be granted together. A fixed collection of transactions are assumed to cycle continuously through the system. Transactions request all needed locks at the beginning. There is no cost for deadlock detection and transaction restart. *well lock placement* strategy is used.

A probabilistic lock contention model is used to simulate the conflict of lock requests from concurrent transactions. The lock conflict is computed as follows. There are $lgran$ granules in the database. Suppose there are k active transactions and each transaction T_i has L_i granules locked. Dividing the interval $(0,1]$ into $k+1$ partitions.

$$P_1 = (0, L_1/lgran]$$

$$P_2 = (L_1/lgran, (L_1 + L_2)/lgran]$$

.

.

.

$$P_i = (sum(L_j, j < i)/lgran, sum(L_j, j \leq i)/lgran]$$

$$P_{k+1} = (sum(L_j, j \leq k)/lgran, 1]$$

Lock conflict is calculated by choosing a random number p uniformly distributed on $(0,1)$. If p is in P_j for some $j \leq k$, then the transaction is blocked by the i th transaction. Otherwise it may proceed. This paper assumes that enough granules are unlocked for the requesting transaction to potentially proceed, thus the data contention is very low.

Lock table is assumed to be kept in the second storage. Lock I/O and CPU time are given as input parameters. Under these assumptions, they run different experiments by varying different parameters. They conclude that small number of

granules, for example, for database with 5000 entities, 10 granules is sufficient.

Dynamic locking in centralized database:

Ries and Stonebraker extended their locking granularity study of [25] using the same model [26]. The main modifications in this study are:

- Dynamic locking is applied.
- In addition to the *well lock placement*, *random lock placement* and *worst lock placement* are also examined.
- Lock table that is maintained in main memory is considered.

Under the new assumptions, lock overhead for fine granularity and small transactions has less impact on system performance than that in the last study. Fine granularity is preferred for applications only with small size transactions accessing the database randomly. For large transactions, coarse granularity is still the most efficient. This study assumes that deadlocks are rare [25].

Static locking in parallel database:

The impact of locking granularity on parallel database system performance was studied by Dandamudi, Au and Chow in [9] by applying the static locking scheme. Transaction and sub-transaction scheduling policies were also studied.

They apply the simulation model extended from the model in [25]. Instead of one processing node, a transaction is split into sub-transactions that execute independently in multiple processing nodes. All three lock placement strategies discussed above are examined in this paper. Their results support the conclusions in the study [26] even for large parallel database systems. Besides, for the same workload, finer granularity is desired with increasing system size when the database is accessed sequentially. When the system is heavily loaded, coarse granularity is desirable. They conclude that providing granularity at block and file level may be adequate for practical purpose.

Chapter 3

Simulation Design and Experimental Environment

3.1 Database Model

Database model captures the characteristics of the database, such as the database size, data distribution on multiple nodes, and access pattern etc.

In this study, database is modeled as a collection of relations. Each relation contains the same number of records. Several records are grouped in a page. When accessing the disks, the resource manager reads one block of data into the main memory. Usually one block can hold the amount of data of one page. Hence the number of records in a page is determined by the size of a record (*rdSize*) and a block (*blockSize*) as a number of bytes.

$$\text{Number of records in a page} = \left\lfloor \frac{\text{blockSize}}{\text{rdSize}} \right\rfloor$$

Database size (*dbSize*) is defined as the total number of records in the database. It can be derived from the number of relations (*numOfRels*) in the database and the number of records in each relation (*relSize*).

$$\text{dbSize} = \text{relSize} * \text{numOfRels}$$

All relations are horizontally fragmented among all disks using the round-robin

strategy. The degree of fragmentation is at the page level. At each round, a number of pages equal to the number of disks are moved and each page goes to one disk. All records in one page are distributed to one node. Therefore, the number and size of partitions are determined by the total number of nodes in the system and the database size.

The access mode allowed to the database can be either read or write. When a transaction reads the database, it only retrieves the information from the database. When a transaction writes the database, it may update the data. Each lock can be requested with either read or write mode. Three degrees of granularity are studied, which are *record*, *page* and *relation*. *Record* is the finest granularity and *relation* is the coarsest granularity. The number of granules in the database depends on the granularity. If the granularity is *record/page*, the number of granules will be equal to the number of records/pages in the database. Due to the fact that a relation is distributed to multiple nodes, the size of a granule with *relation* granularity is defined as all records of a relation in a node. Therefore, the number of granules is always the same as in a single node for the *relation* granularity.

Lock overhead consists of lock I/O and CPU time. CPU lock time is measured in run time. If the lock table is maintained in the main memory, the I/O time for requesting a lock will be 0. In the experiments where the lock table is in secondary storage, the I/O lock time will be given as an input parameter. We assume 10 locks are grouped into one disk block. Each time a transaction requests a lock, the block containing this lock will be read into the main memory.

The number of transactions in the system (mpl) at one time is another parameter in database modeling. Database Management System (DBMS) allows more than one transaction in the system to improve the system performance, but too many transactions in the system will cause data contention and lead to the poor performance. The appropriate value of mpl should be determined carefully by doing some preliminary experiments.

3.2 Workload Model

Workload model characterizes the workload in the system. The arrival of the transactions to the system and the amount of data to be accessed by each transaction are the most important issues related to the system workload.

Transactions are modeled as a collection of independent requests to the database. Each request is characterized by the data to be accessed and the access mode. In this model, the access mode can be either read or write according to the access pattern allowed in the database model. Therefore, a transaction is modeled as a sequence of operations, each has the form $\langle Read/Write, RelNum, RdNum \rangle$ [34]. *relNum* and *rdNum* indicates the record and the relation the transaction accesses.

The size of a transaction is defined as the number of records to be accessed. It is uniformly distributed over the range of a minimum and a maximum transaction size. The average transaction size is (maximum-minimum)/2.

The accesses of a transaction are distributed uniformly among all records in the database. Data skew is a problem in shared-nothing parallel database systems. To study this problem, normal distribution is used to obtain data skew effect so that transactions access some data more frequently.

The arrival process is generally modeled either as an open system or closed system. In an open model, transactions arrive from outside. In a close model, users circulate through the system and resubmit transactions with a new set of parameter values after the previous one is completed. Therefore the system always has a fixed number of transactions. This study is based on the close model. A number of terminals are simulated as the number of users who generate the workload to the system. Each terminal has a *thinkTime* which is the time between a transaction completion and the next transaction submission from the terminal. The value of *thinkTime* affects the system workload. When the think time is small, terminals generate transactions quickly, and thus the workload is heavier for the system. The *thinkTime* is uniformly distributed between a pre-defined minimum and maximum value for each terminal.

3.3 System Model

The system model characterizes the behavior of the actual parallel database system as shown in Figure 3.1. This simulation model is an extension of the model used in [25].

At the initialization stage, each terminal generates a transaction and submits it to the transaction ready queue. Then all transactions have to go through the following stages.

1. The transaction at the head of the transaction ready queue is removed. It is split into several sub-transactions based on the node that the data to be accessed resides. Each sub-transaction is sent to the appropriate node, via the interconnection network, where its requests can be processed. No two sub-transactions of a transaction will be assigned to the same node. Each sub-transaction requests locks locally and independently.
2. Sub-transaction requests locks. If the locks are granted, the sub-transaction is placed at the end of the I/O queue. If the locks are denied, the sub-transaction is placed at the bottom of the blocked queue. System records the blocked transaction in order to resume it after its blocking transactions finish processing and release their locks.
3. After completing the I/O, the sub-transaction is placed at the end of a CPU queue.
4. After the completion of the required CPU processing, the transaction goes back to step 2 to request the next lock and restarts steps 3 and 4 till it finishes all requests.
5. After the sub-transaction finishes all requests, it waits for other sub-transactions of its parent transaction to complete. A transaction can complete only when all its sub-transactions are completed.

6. A transaction can be either committed or aborted. In this simulation, a transaction may be aborted only because of a deadlock, either a local deadlock or a global deadlock. In both cases, the transaction will release all locks held by its sub-transactions. If the transaction is committed, it finishes and exits the system and a new transaction will enter the system after a think time. If the transaction is aborted, it will be appended to the end of transaction ready queue.
7. When a transaction releases its locks, the blocked transactions waiting for it will be removed from the blocked queue and put at the front of concurrency control queue.

3.4 Transaction Processing System Design

3.4.1 Overview

A transaction processing (TP) system is a computer system that processes transaction programs and provides tools to ease or automate application programming, execution, and administration. Basically, a TP system includes application generators, operation tools, one or more database systems, and networking as well as operating system software [17].

Concurrently, client/server architecture is popular for designing a TP system. In client/server environment, a large number of computers communicate with shared servers. The client calls the server by either remote procedure call (RPC) or peer-to-peer communication. The server processes the requests of its client by using the underlying resources and returns the result to its client. There is a similarity between the client/sever and TP environment. A TP system consists of a large number of display devices, called terminals. These terminals are connected to shared servers that run transactions. Therefore, a TP system is naturally and commonly designed by applying the client/server architecture. A common configuration is to have the

Figure 3.1: System Model

client running on a desktop machine and the server on a larger machine. In some cases, the client runs on a shared server, which calls another server.

TP system design involves many considerations such as the transaction concurrency control and system recovery. Since this study is concerned about the concurrency control, the design of the system also focuses on this aspect. We assume that the system is reliable, thereby no system failure and recovery are handled in this design. We adopt the client/server architecture. The client processes, where the transactions are generated, makes requests to server processes running on other computers. All requests submitted by clients and results obtained by servers are sent by passing messages. There are two types of nodes in this design, host node and processing node. The database is partitioned horizontally among all processing nodes. There are three processes residing in the host node. Transactions are generated by transaction generator (TG) process. The transaction manager (TM) process communicates with the TG and directs the execution of clients requests. Another process, called global deadlock detector (GDD), is responsible for detecting the global deadlock when there is more than one processing node in the system. Transactions are executed on the processing nodes. Each processing node contains 4 processes, cohort process, which executes on behalf of a sub-transaction; lock manager (LKM) process, which processes the lock requests; I/O and CPU processes, more like a resource manager, which represent and manage the cohorts to access the disks and CPU.

Figure 3.2 shows different processes and their responsibilities. The next subsections describe the design of each module in more detail as well as the main algorithms used in the system.

3.4.2 Transaction Generation

All transactions are generated in the transaction generator (TG) process. To simulate that the transactions are from different users, several terminals representing the end users are created in this process. The TG process keeps the track of all terminals and ensures that each terminal behaves independently. Terminals generate

TG -- Transaction Generator

LKM -- Lock Manager

TM -- Transaction Manager

GDD -- Global Deadlock Detector

Figure 3.2: Transaction Processing System Design

transactions independently by using different seed values in the distribution function for choosing all parameters related to the transaction generation. The transaction size, in terms of number of records, and the number of relations to be accessed are determined first. The average transaction size is given as an input parameter. Then for each record, the lock mode, relation name and record name are chosen in a loop. Since there are more than one terminal, TG can not really think and stop running, instead it simulates the *thinkTime* of each terminal. TG maintains a table containing the information about the *thinkTime* and the most recent transaction completion time for each terminal. TG keeps checking the table after a very short period. When it is the time for a terminal to generate the next transaction, TG informs the terminal, then replaces the *thinkTime* with a new value.

TG only communicates with TM process. It sends a new transaction to TM and receives either an error message if there is an error with the data generated or the completion message when its transaction completes. If an error message is received, the terminal will generate a new transaction immediately.

3.4.3 Transaction Management

Transaction manager (TM) manages and coordinates the flow of transactions. Each transaction executed in the system is supervised by a single TM, meaning that the transaction issues all of its database operations to that TM. There are several responsibilities that the TM has to perform correctly.

- [Transaction acceptance and execution]

Once a new transaction enters the system, TM checks the availability of the data to be accessed first. TM has the information about the location of the data in the system. If one transaction tries to access a data that does not exist in the database, TM will return an error message to inform the TG. If the transaction passes the validation, TM will choose the execution nodes for it and split the transaction to several sub-transactions. All sub-transactions will be sent to the processing nodes in the system. Then TM launches one

cohort process for each sub-transaction to perform the requested operations against the database in the appropriate processing node.

- [Transaction commit]

Each cohort executes at its own processing node independently. All cohorts are coordinated and synchronized by the TM through communication. TM models the detail of a two-phase commit protocol (2PC) that ensures the atomicity property of a transaction. All cohorts of the transaction are either committed or aborted. In this implementation, all cohorts belonging to one transaction join one group after they are loaded by the TM. TM records their group name so that it can be used later to broadcast commit or abort message. Since the system is assumed to be reliable, there is no system failure and recovery. A transaction is restarted only when it causes a deadlock. Any cohort can initiate the 2PC protocol. The basic 2PC is adjusted from both TM and cohort views and shown in the next page.

TM may receive a vote message from a cohort after it has already aborted this transaction due to the delay of communication. This situation will not happen if the transaction is committed since the TM commits the transaction only after it receives the votes from all cohorts. To handle this problem, TM maintains a list for active transactions. Every time the TM gets a vote request, it checks if the sender is in the active list or not. If it is not, the TM knows that the transaction has been already aborted and will just ignore the vote message.

Centralized Two-phase Commit Protocol

At transaction manager (TM):

1. If “VOTE_COMMIT” is received from all cohorts of a transaction, the TM broadcasts a “COMMIT” message to all cohorts in the group. At anytime, if the TM receives a “VOTE_ABORT” message from a cohort of a certain group, it broadcasts an “ABORT” message within that group.
2. Waits until the acknowledgment is received from all the cohorts in the group, either “COMMITTED” or “ABORTED”. Then it deletes all information about the transaction.

At a cohort:

1. After finishing the execution of its portion of a transaction, it sends a “VOTE_COMMIT” message to the TM. A cohort may send a “VOTE_ABORT” message to the TM if it causes a deadlock and is chosen as the victim.
2. If a “COMMIT” message is received, the cohort releases all locks and sends back a “COMMITTED” message to the TM. Otherwise, if a “ABORT” message is received, it releases all locks and sends the TM an “ABORTED” message.

Even when there is no cohort voting for abort, it is possible to abort a transaction. When there is a global deadlock, the victim will be chosen by the global deadlock detector and sent to the TM. In this case, it is the TM’s responsibility to abort the victim. TM will simply broadcast an “ABORT” message to all the cohorts of the deadlocked transaction, then follow the other steps in the 2PC described above.

- [Multiprogramming level control]

The multiprogramming level (*mpl*) refers to the number of active transactions allowed in the system. If too many transactions are allowed in the system, a high level of data contention environment may result in poor system performance. One reason for this is the lock waiting time. When the data contention is high, allowing a new transaction into the system actually causes more transactions waiting for the locks held by others. The number of runnable transactions decreases. The other reason is the abortion of transactions. Admitting a new transaction increases the possibility of a deadlock. A transaction that executes for some time and is later aborted leads to wasted resource capacity.

Data contention is determined by the multiprogramming level, database size, transaction size and transaction service time etc. In this study, database size is fixed, transaction size and service time are generated by the transaction generator, and the multiprogramming level is controlled by the TM by accepting or refusing a new transaction to the system from the transaction ready queue.

3.4.4 Lock Management

Lock Manager (LKM) is responsible for monitoring the accesses of concurrent transactions to the shared database. Both distributed lock manager (DLKM) and centralized lock manager (CLKM) are implemented. The main difference between a DLKM and a CLKM is where they are located. DLKM resides in the processing node, while CLKM is placed in the host node. Each DLKM has its own lock table and grants requests locally, while the CLKM only maintains one lock table and handles the lock requests from sub-transactions at different nodes. With DLKM, a global deadlock detector is required to detect the global deadlock. With CLKM, both the local and global deadlock will be detected by the CLKM whenever there is a block. The basic functionality and algorithms employed in these two lock managers are the same. The following discussion is based on DLKM.

The lock table has an entry for each lock which can be a *record*, a *page*, or a

relation. Since each relation is partitioned and distributed to different processors evenly, with the granularity of *relation*, each entry in the lock table refers to a part of a relation. Lock mode can be either write or read. Each lock has a list, *lock list*, containing the information about the transactions that are either holding or waiting for this lock and their requested lock mode. A lock may be granted to more than one transaction. A new lock request can be granted only if its requested lock mode is compatible with all the granted transactions. Therefore, each lock is also assigned a *group mode* which is the highest lock mode among all granted transactions. For instance, write mode is higher than read mode. The group mode is updated each time the DLKM grants a request. The DLKM also maintains a *lock hold list*, which records all active transactions that have been granted a lock and all locks they are holding, and a *wait list*, which contains the transactions who are waiting. These two lists are used for releasing locks and detecting deadlocks.

All DLKM performs the same responsibilities, mainly granting and releasing locks and detecting local deadlocks.

- [Lock granting]

DLKM processes lock requests in a FIFO fashion. When a lock request arrives, DLKM goes to the lock table and checks the requested lock status first. To avoid starvation problem in which a request will never be granted, we adopt the policy that if there is any lock request in the waiting list of a lock, other new requests to this lock have to wait even though their requested lock mode may be compatible with the lock group mode. The lock request can be either a new request or a conversion request. Lock conversion is allowed when a lock is requested by a transaction that has already held the lock. For example, when a transaction requests a read lock on some data, then it wants to write the data, the lock can be converted to write. Conversion request has higher priority than new request because we want the active transaction to finish and release its locks as soon as possible so that more transactions can request locks. Therefore, when there is a lock conflict, the transaction that wants to convert

the lock will be granted before the transaction that requests a new lock. The following is the granting algorithm implemented in the DLKM.

Request Granting Algorithm

Begin

If lock request is a converting request

If lock request is compatible with group mode

grant the request and modify the group mode

Else block the request and check for a local deadlock

Else lock request is a new request

If no transaction is waiting for the lock &

lock request is compatible with group mode

grant the request and modify the group mode

Else block the request and check for a local deadlock

end

- [Lock releasing]

Releasing the locks of a transaction involves deleting all records related to the transaction in all lists and tables maintained by the DLKM and granting the waiting requests blocked by this transaction. DLKM performs lock release function when a transaction completes or is restarted because of deadlock.

- [Local deadlock detection]

Whenever a lock request is blocked, the DLKM has to detect the local deadlock by checking if there is any cycle in the wait-for graph (WFG). Figure 3.3 shows how the DLKM derives the WFG from the transaction *wait list* and *lock lists*. The *wait list* shows that transactions t1, t2 and t3 are blocked and waiting. In the *lock list*, the non-shaded area indicates the granted transactions with the granted mode, and the shaded area indicates the waiting transactions with

its request mode for this lock. For example, the first *lock list* shows that transaction t_2 is holding a write lock on this lock, while transaction t_1 is waiting for this lock with a read mode. The WFG is constructed and shows that there is a cycle between t_1 and t_2 . Either t_1 or t_2 has to be aborted and restarted. The deadlock detection algorithm is given at the next page [34].

Sometimes, there are more than one cycle in the WFG. To minimize the effect of a deadlock, DLKM is designed to check the WFG more than once until there is no cycle in the WFG.

Figure 3.3: Data Structures of wait list, lock list and WFG

Deadlock Detection Algorithm [34]

Input: A linked list representing WFG containing: wait transactions, locks held by each transaction and the wait-for transactions.

Data Structure Used: A FIFO stack containing the deadlocked transactions.

Output: Whether there is a deadlock or not, if so, the victim.

Begin

 Initialize the stack to empty

for next transaction in the adjacent linked list

begin

 Push the next transaction into the stack

while stack is not empty

if next waiting-for of top transaction in the stack exists

begin

 T-next=next transaction-waiting-for

if T-next exists in WFG

if T-next transaction is in the stack

 there is a deadlock & return the victim

else

 push T-next to the stack

end

else

 pop stack

endWhile

endFor

 return NULL

End

3.4.5 Global Deadlock Detection

Global deadlock is a typical problem with shared-nothing systems where transactions execute on more than one node independently and only are synchronized at the commit stage.

The detection of global deadlock is the responsibility of the global deadlock detector (GDD). Periodically, the GDD sends messages to all DLKMs to collect their local WFGs, then it combines them into a global WFG and detects cycle by applying the same algorithm described before. Each time the GDD detects a deadlock, it sends the victim to the TM, and then modifies the WFG and continues until there is no cycle in the WFG.

3.4.6 Resource Management

Each time a transaction gets a lock, it will go through an I/O phase to read the data and a CPU phase to process the data. There are two processes running to handle and manage the I/O and CPU respectively. Both I/O and CPU processes apply the FIFO policy. I/O process simulates the disk access. Each time the I/O process accesses the disk, it brings a page into the main memory. If a transaction wants to access a page that is already in the main memory, the I/O time will be 0. To know which page is in the main memory, the I/O process maintains a list that has an entry for each page. Whenever there is a I/O request coming, the I/O process checks the status of the requested page first. If the page is not in the main memory, the I/O process will wait for an I/O time and then respond. Otherwise, it will respond immediately. The CPU process simply simulates a CPU time for all CPU requests. The amount of I/O and CPU required by a transaction are determined by the number of records accessed by it and which pages are in the main memory.

3.4.7 Network Management

In a shared-nothing database system, processors communicate by sending messages. In some studies, the communication is simulated by a network manager. In this

study, processors actually send messages and the communication is handled by the software package PVM. A task can send a message to another task by simply calling a pvm function. The other task can receive the message either synchronously or asynchronously. The network used is an Ethernet with 10Mb/s. The network is assumed to be perfectly reliable: if node A sends a message to node B, node B is guaranteed to receive the message without error. The details of the PVM are discussed next.

3.5 Experimental Environment

All experiments in this study are run on Triton, a NOW-based parallel system. Triton nodes communicate by passing messages explicitly. PVM software package is used for this purpose.

3.5.1 NOW-based Parallel System

Shared-nothing database system is simulated in this study by utilizing the available NOW-based parallel system, called Triton. Triton is composed of a number of Pentium processors linked with 10Mb/s and 100Mb/s Ethernet networks. There are currently 6 Pentium and 4 Pentium pro nodes in the system. The main node known as Triton from the outside is a P6/200 with 32Mb of RAM and a hard disk. This is the only node that can be accessed from the outside, and the only node that supports X11. The other nodes are diskless.

Triton is a research system. In order to facilitate some critical experiments, all nodes, except Triton, can work in either a *shared* mode or a *non-shared* mode. *non-shared* mode is realized by allocating the nodes to the user. Since networked computers can have several other users running a variety of jobs, the machine load can vary dramatically. The time for sending a message can also vary depending on the network load imposed by all the other network users. To avoid this problem, *non-shared* mode is used when running experiments.

3.5.2 Parallel Virtual Machine (PVM)

Parallel Virtual Machine (PVM) is a software package that permits a heterogeneous collection of computers hooked together by a network to be used as a single large parallel virtual machine. Thus large computational problems can be solved cost effectively by using the aggregate CPU power and memory of several computers. PVM is portable and runs on a wide variety of modern platforms such as 386/486 PC class machines, high-performance workstations, and shared-memory multiprocessors, etc. Figure 3.4 shows the architecture of PVM [23].

One of the features of PVM is heterogeneity. PVM supports heterogeneity at the application, machine, and network level. The networked computers can have different architecture, data format, computational speed, machine load and network load. PVM handles all data conversion that may be required if two computers use different integer or floating point representations.

PVM has two parts:

- A daemon process called *pvm* that resides on all computers making up the virtual machine. Any user with a valid login can install this daemon on a machine.
- A user library that contains routines for initializing tasks on other machines, communication and changing configuration of the PVM machine.

The PVM system currently supports C, C++ and Fortran languages. A task in PVM is defined as a unit of computation analogous to a UNIX process. All PVM tasks are identified by an integer task identifier (*tid*). Virtual machine allows the tasks to communicate and synchronize with each other by sending messages. Messages are sent and received by using *tids*. Since *tids* must be unique across the entire virtual machine, they are supplied by the local *pvm*. The PVM application can then be started from a system prompt on any of the hosts.

PVM provides the facility to group several tasks together. When a task joins a group, it is assigned a unique “instance” number in that group. Any task can

Figure 3.4: PVM Architectural Overview

broadcast messages to the group. In this study, all sub-transactions belonging to one transaction are assigned to one group and coordinated by the transaction manager.

PVM supplies functions to automatically start up tasks on the virtual machine. By default, when PVM spawns a task, it tries to spawn it on the least loaded machine. User may also specify the node. Since shared-nothing database system is modeled in this study, no two sub-transactions can be sent to the same processor. The node where a sub-transaction should go is specified.

PVM provides a certain level of fault tolerance. There are several reasons for PVM process failure, such as not enough memory, system crash, or machine is reboot. When a failure occurs, PVM will notify users of the failure. Users may specify the notification message in the program, then they can know where the problem is.

Chapter 4

Performance Evaluation

Performance is a critical aspect of a transaction processing (TP) system. The *response time* to end users is a very important measure of TP system performance. Besides the response time, companies who use TP system also want to process as many transactions as possible. This aspect of system performance can be measured by system *throughput*. There are also some other factors that reflect the system performance and are described in this chapters.

4.1 Simulation Parameter Settings

The purpose of the simulation is to explore the desired size of a granule and the impacts of different factors on the system performance and the choice of locking granularity. Experiments were conducted by varying different parameters such as the number of transactions allowed in the system, the average size of transactions, the system size, and the think time of terminals. Transactions are generated in the host node and processed in the processing nodes. The accesses of transactions are initially uniformly distributed among all records in the database. The problem of data skew in shared-nothing parallel database system is examined by modifying the distribution of access to normal distribution. Experiments were conducted with different degree of data skew. Distributed lock manager (DLKM) and centralized lock manager (CLKM) are examined and compared in this study. Each experiment

is run for three degrees of granularity, which are *record*, *page*, and *relation*. *Record* is the finest granularity, while *relation* is the coarsest granularity. Lock table is assumed to be either in the main memory or on the disk. When the lock table is in the main memory, the lock overhead has less impact on the system performance because there is no lock I/O overhead involved.

All experiments are run for 31 batches. 95% *Confidence interval* is calculated and kept less than 3% to ensure the stability and accuracy of the result of each batch.

The default input parameters required to drive the simulation are listed in Table 4.1. Experiments are run by varying some of these default parameters, such as the *mpl*, the transaction size, the number of processing nodes, the think time and the degree of data skew. The other input parameters are fixed for all experiments. The deadlock detection period is chosen after doing several preliminary experiments. We didn't use continuous deadlock detection scheme mainly due to the communication cost involved. At the same time, we also don't want to let the deadlocked transactions staying too long in the system. The other parameters, such as the lock I/O time, I/O time for reading and CPU time for processing are chosen the same as those in the previous studies. The values of these parameters are considered as the reasonable values.

In this scenario, 64 terminals submit transactions to the database, which consists of 16 relations each having 4000 records. The *blockSize* and *rdSize* are fixed at 1000 and 200 bytes respectively, thus each page has 5 records. Database is partitioned and distributed to the number of *numOfProc* nodes. Only one transaction manager is used and thus the number of host node is set to 1.

Data contention is studied in this simulation. The probability of deadlock is not as small as that in the previous studies. On the other hand, we don't want too many deadlocks. This is because that the impact of the overhead resulting from deadlocks may dominate the impact of lock overhead and affect the investigation of locking granularity. Larger *mpl* will cause more deadlocks. After running several preliminary experiments, the value of *mpl* are set to 2 and 3 for the experiments

Parameters	Meaning	Value
------------	---------	-------

Database Model Parameters

<i>rdSize</i>	Number of bytes of one record	200 bytes
<i>blockSize</i>	Number of bytes in one block	1000 bytes
<i>relSize</i>	Number of records per relation	4000 records
<i>numOfRels</i>	Number of relations in the database	16
<i>dbSize</i>	Number of records in the database	64000 records
<i>numOfHost</i>	Number of host nodes	1
<i>numOfProc</i>	Number of processing nodes	8
<i>mpl</i>	Number of transactions allowed in the system	2

Workload Model Parameters

<i>numOfTerms</i>	Number of terminals	64
<i>thinkTime</i>	Think time of terminals, uniformly distributed	0..10 sec
<i>tSize</i>	Average transaction size as a percentage of database	0.5 %
<i>writePro</i>	Write probability for update-based transactions	1/4

Other Parameters

<i>ioTime</i>	I/O time for accessing one page	0.02 sec
<i>cpuTime</i>	CPU time for processing one record	0.005 sec
<i>lkIoTime</i>	lock I/O time for reading one lock	0.02 sec
<i>detTime</i>	Global deadlock detection period	1 sec

Table 4.1: Default Simulation Parameter Settings

Parameters	Meaning
RT	Mean response time
QT	Mean queuing time
TP	Average throughput of system
LK	Mean lock overhead
BL	Total number of blockings
LD	Total number of local deadlocks
GD	Total number of global deadlocks
CT	Mean communication overhead

Table 4.2: Output Statistics

where the lock table is in main memory and disks respectively. 2 is chosen for the experiments with lock table in main memory because the lock overhead is less and has less impact.

The value of I/O time is set as four times of the CPU time to simulate the “I/O bound” applications. If the lock table is on disk, the lock I/O time will be equal to the I/O time, otherwise, it will be 0. The value of global deadlock detection period *detTime* is fixed at 1 second after trying different values of detection period. This value is set small to make sure that deadlocks can be detected and resolved as soon as possible. At the same time, the cost of communication involved in deadlock detection needs to be considered. Therefore continuous detection is not the choice.

Table 4.2 shows the output statistics obtained in each simulation run. *Response time* is calculated from the time that a transaction is sent by a terminal to the time the transaction is completed. *Queuing time*, which is a portion of the response time, represents the waiting time of a transaction in the transaction ready queue before it is processed. *Throughput* is defined as the number of completed transactions per unit time. The *response time* and *throughput* are the most important parameters reflecting the system performance. Other parameters will help evaluate and under-

stand the system performance from different aspects. *Lock overhead* for requesting, setting and releasing locks shows how the locking granularity and thus the number of locks in the database affect the system performance. *blocking ratio*, which indicates the frequency of a transaction is blocked, is calculated by dividing the *number of blockings* in one node by the number of transaction completed. The number of deadlocks are recorded by the number of *global deadlocks* and the number of *local deadlocks* separately. The *abort ratio* is calculated by the fraction of the total number of aborts and the completed transactions to show how often a transaction be restarted. In shared-nothing database systems, processors communicate by sending messages. *Communication overhead* reflects the impact of the communication cost.

The following sections discuss the results obtained from the experiments. For each experiment, we provide five plots, which are the *response time*, *throughput*, *lock overhead*, *abort ratio* and *blocking ratio*. Since the communication overhead is small and dependent on the dynamic workload of the system, we will not discuss the impact of communication.

4.2 Validation of the Model

This section describes the validation tests we performed to ensure the correctness of the simulation model. There are several output parameters we collected to help the validation. The lock manager records the lock CPU time for one record and total lock overhead during the run. The I/O process records the total number of requests and the number of real I/O reads. The transaction manager records the queuing time and also accumulates the communication and implementation overheads that are recorded and submitted by other processes at the end of run for each batch. Implementation overhead mainly includes the time spent in preparing the outgoing messages and unpacking the received messages.

First, we conducted tests in which there is no lock conflict among transactions. Without a lock conflict, the response time will depend mainly on the queuing time, the I/O and CPU processing time, and the lock overhead. Communication and

implementation overhead will affect the response time only marginally. We executed one transaction on one node and then executed 10 transactions submitted by one terminal on one node. In this case, there is no queuing time because a single terminal always submits a new transaction after the previous one is completed. The I/O time and CPU time for each request are the input parameters and we know the number of real I/O read operations. We also know the approximate lock overhead. Therefore, we can calculate the response time. The result we got from running the test is very close to the value we calculated. We also conducted a validation test that executed 10 transactions submitted from different terminals with a *mpl* of 1. The mean response time increased reasonably. This proves that our model works correctly when running transactions on one node sequentially. Then we conducted the same tests on 2 and 4 processing nodes. The response time decreases almost proportionally. This is also reasonable.

Second, we did tests with lock conflict. Since lock conflict is handled by the lock manager dynamically, we can't validate the result as precisely as we did in the first set of experiments. We executed one, two and ten transactions by varying different parameters, such as the *mpl* and the lock I/O time. The results we got are within the range of the predicted values.

4.3 Performance with Distributed Lock Manager

This section discusses the performance of DLKM. The first subsection shows the impact of *mpl* on the system performance. The lock table is assumed to be in main memory. This subsection presents and discusses all five plots mentioned in the last section. Since the impact of different parameters on system throughput is the same as that in system response time, the plots for the throughput for the remaining subsections are moved to the appendix. For other subsections, the results are collected for two cases. One is when the lock table is on disk. The other is when the lock table is in the main memory. We discuss the case where lock table is on disk first. For the other case, if the plot shows the similar behavior with the first

Figure 4.1: Response time with DLKM (lock table in main memory)

case, we will mention it and move the plot to the appendix.

4.3.1 Sensitivity to mpl

The value of mpl represents the number of active transactions allowed in the system at the same time in a multi-user database system. Theoretically, larger mpl allows higher degree of concurrency. But at some point, allowing one more transaction into the system actually decreases the number of runnable transactions and thus leads to poor system performance. In this set of experiments, the value of mpl is set at 1, 2, 3, 4 and 5. Lock table is assumed in the main memory. The other parameters are kept at the default value.

Figure 4.1 shows the impact of mpl on the system response time. For a fixed granularity, increasing the mpl decreases the response time. The best performance is reached at a mpl of 3. Further increases in mpl increases the response time. The improvement in the response time is due to the factor that with larger mpl , more transactions can request locks simultaneously and thus the transaction waiting time

Figure 4.2: Blocking ratio with DLKM (lock table in main memory)

Figure 4.3: Abort ratio with DLKM (lock table in main memory)

Figure 4.4: Lock overhead with DLKM (lock table in main memory)

and processor idle time are decreased. For example, when the mpl is 1, transactions will run sequentially. This causes all other transactions to wait even though some processors are idling. On the other hand, the lock contention increases when more transactions are in the system. The increase in the response time after mpl of 3 is the result of two factors. One is the lock waiting. With high lock contention, allowing a new transaction into the system actually causes more transactions waiting for the locks and more blocking overhead associated with each transaction. The other factor is the increase in deadlocks due to the same reason. A transaction that executes for some time and is later aborted leads to wasted resource capacity. The impacts of mpl on the blocking ratio and abort ratio are shown in Figures 4.2 and 4.3. For a fixed granularity, both the blocking ratio and abort ratio increase with increasing mpl . When the blocking and deadlock overhead outweigh the advantage of providing higher concurrency, the system performance deteriorates.

The increase in deadlocks also affects the lock overhead because more time is spent on locking and unlocking the deadlocked transactions. The increase in lock

Figure 4.5: Throughput with DLKM (lock table in main memory)

overhead is shown in Figure 4.4. Furthermore, this figure also shows that the degree of increasing lock overhead for *page* granularity is much higher than that for the other two granularity. This is because the abort ratio of *page* is higher. From Figure 4.3, we observe that the abort ratio of *page* is the largest, followed by *record*, and *relation*. Recall that the deadlocks occur when transactions are blocked and wait for each other. It is interesting to note that the blocking ratio of *page* is less than that of the *relation* and the *record* has the least blocking ratio. This disparity between the blocking ratio and abort ratio is due to the fact that the lock contention of *relation* is very high. In such high contention environment, transactions are blocked at the beginning when they enter the system before holding many locks, and thus cause fewer deadlocks. Certainly, if the lock contention is very low, such as the *record*, deadlocks will be also less frequent. This is why the *page* has the largest abort ratio. The abort ratio of *record* is higher than *relation* for the same reasons.

Figure 4.5 shows the impact of *mpl* on system throughput. For the reasons discussed above, the system throughput increases with increasing *mpl* and then

Figure 4.6: Response time with DLKM (lock table on disk)

decreases after the mpl is larger 3.

From the above discussion, we know that choosing an appropriate value of mpl is important in the design of a database management system. Furthermore, fine granularity may cause more deadlocks. For the model and assumptions used in this study, we conclude that *page* may cause more deadlocks than *record* and *relation*.

4.3.2 Sensitivity to Transaction Size

To study the impact of transaction size on the desired locking granularity, the number of records to be accessed by a transaction is varied from 0.05% to 1% of the database. Lock table is assumed to be in either main memory or on disk.

Lock Table on Disk

Figure 4.6 shows the impact of transaction size on the system response time. For a fixed granularity, with increasing transaction size, the response time increases because more requests and thus more execution time are involved with each transac-

Figure 4.7: Lock overhead with DLKM (lock table on disk)

tion. Among the three degrees of granularity, the *relation* has the best performance for all different transaction size. The finest granularity, *record*, has the highest response time. The choice of granularity is affected by the trade-off between the lock overhead and the allowed degree of concurrency. Fine granularity allows higher degree of parallelism and thus more transactions can be executed concurrently. The disadvantage of fine granularity is more lock overhead associated with locking and unlocking operations. In a high data contention environment, the deadlock overhead also affects the desired granularity. The impact of transaction size on lock overhead and abort ratio are shown in Figures 4.7 and 4.8. The lock overhead of *record* is much larger than that of *relation*, especially with larger size of transactions. The *page* can not get the better response time than *relation* is because that the total overhead of locking operations and deadlock of *page* outweighs the advantage of increased concurrency provided by *page*.

In Figure 4.8, for small size transactions, such as less than 0.1%, the abort ratio of *relation* is the largest. Then as the transaction size increases, the abort ratio of

Figure 4.8: Abort ratio with DLKM (lock table on disk)

Figure 4.9: Blocking ratio with DLKM (lock table on disk)

page and *record* start to increase while that of *relation* becomes flat and finally goes down. For small size transactions and fine granularity, the probability of lock conflict is very small. Therefore, transactions can potentially acquire their locks, and there are fewer blockings and deadlocks. As the transaction size increases, more locks are held by each transaction and the lock contention increases. The reason for the variation of abort ratio with increasing the transaction size is explained in the last section. With very high lock contention, coarse granularity may cause fewer deadlocks than fine granularity. The impact of transaction size on blocking ratio is indicated in Figure 4.9. The blocking ratio for each granularity increases with increasing transaction size because of the increasing lock contention. When the size of transaction is very large, the blocking ratio of *relation* decreases and that of *page* tends to be flat. This is because, with very high lock contention, the transactions have less chance to get a lock and will wait till the blocking transactions are completed. In this case, the transactions are blocked less and longer each time. Since coarse granularity causes higher contention, the blocking ratio of *relation* goes down. This also shows why coarse granularity has fewer deadlocks than fine granularity in a high data contention environment.

Lock Table in Main Memory

Due to the large range of values and the closeness of the results for the three degrees of granularity, the results are divided into small and large size transactions. The average number of records accessed by small transactions ranged from 0.01% to 0.1%. Transactions which access 0.3% - 1% records are considered large size transactions. This distinction, which is arbitrary, is used only for convenience in presenting the data.

Figures 4.10 and 4.11 show the impact of small and large transactions on the response time with different degrees of granularity. In contrast to the case where the lock table is on disk, for small size transactions, *record* granularity gives marginally better performance. For large size transactions, the performance at the three degrees of granularity is very close. This is due to the assumption that the lock table is

Figure 4.10: Response time with DLKM—small transaction (lock table in main memory)

Figure 4.11: Response time with DLKM—large transaction (lock table in main memory)

maintained in the main memory. When the lock table is kept in the main memory, there is only lock CPU overhead involved, thereby the lock overhead has less impact on the system performance. Since smaller size transactions request fewer locks, the lock overhead of each transaction and also the difference between fine and coarse granularity are very small. For example, suppose that a sequence of transactions, each only asking for one record, arrive at the system. The lock overhead for all level granularity will be the same, but *record* allows more transactions to request the locks simultaneously. Therefore the *record* will have the best performance. For large transactions, fine granularity does not improve the degree of concurrency much because more records are required by each transaction. Large transactions also request more locks. In this case, the benefit of increasing concurrency of fine granularity is outweighed by the cost of setting additional locks required by each transaction.

The impact of lock overhead, blocking ratio and abort ratio are very similar to those in the last subsection when the lock table is kept on disk except for the scale of value. These plots are given in the appendix.

From the discussion in this section, we conclude that if the lock table is maintained on disk, coarse granularity such as *relation* gives the best performance; if the lock table is in main memory, fine granularity such as *record* is marginally better when only very small size transactions access the database. For large and complex transactions, there is no desired granularity. All three granularities give the similar performance. Besides, there are two cases where the abort ratio can be small. One is when the lock contention is **very** low, the other is when the lock contention is **very** high. Deadlocks may occur more often between these two extreme cases.

4.3.3 Sensitivity to System Size

System size in this study refers to the number of processing nodes in the system. Experiments are run on 1, 2, 4 and 8 processing nodes. The transaction size is set at 0.5% of the database.

Figure 4.12: Response time with DLKM (lock table on disk)

Lock Table on Disk

Figure 4.12 shows the impact of system size on locking granularity and the system response time. For a fixed granularity, the mean response time decreases with increasing number of processors. The reduction of response time is mainly because of the decrease in execution time and lock overhead of each transaction. In shared-nothing database system, a transaction is split into several sub-transactions and executes on different processors concurrently. Thus with larger number of processors, smaller sub-transactions are run on each processor, which results in smaller execution and queuing times. In addition, the lock overhead for requesting, setting, and releasing locks is shared by all processors. Therefore the overhead placed on each processor is reduced with larger number of processors. This fact is reflected in Figure 4.13, which shows the impact of number of processors on the lock overhead. Among the three degrees of granularity, *Relation* still gives the best performance.

Figures 4.14 and 4.15 show the impacts of number of processors on the abort ratio and blocking ratio, respectively. The blocking ratio of *page* and *record* decreases

Figure 4.13: Lock overhead with DLKM, (lock table on disk)

Figure 4.14: Abort ratio with DLKM (lock table on disk)

Figure 4.15: Blocking ratio with DLKM (lock table on disk)

with the increasing number of processors. This is because the number of available locks is distributed to different processors and the number of blocks is shared by all processors in the system. The blocking ratio of *relation* does not decrease because the number of locks located in each processor does not change with increasing number of processors. A lock at *relation* granularity refers to all records in that relation in a node. The abort ratio for each granularity increases with increasing number of processing nodes because of the introduction of global deadlocks. Global deadlock occurs when transactions are processed on more than one node.

Lock Table in Main Memory

Except the response time and abort ratio, the other plots for main memory lock table are similar to case where the lock table is on disk. Figures 4.16 and 4.17 indicate the impact of system size on the response time and abort ratio when the lock table is in main memory. For a fixed granularity, the response time decreases like when the lock table is on disk, but the response time for all degrees of granularity is very

Figure 4.16: Response time with DLKM (lock table in main memory)

close. This observation supports the conclusion that when lock table is in main memory, there is no obviously desired granularity for large size transactions.

The abort ratio of *page* and *record*, instead of increasing with the number of nodes, increases first and then decreases. This is because, with the lock table in main memory, the execution time for each transaction is smaller. With a large number of processors, smaller transactions execute in a local processor and there is less time for a transaction staying in the system. Therefore there is less contention and the probability of local and global deadlock is small. If lock contention is not small enough, more global deadlocks occur. This is also the reason that the abort ratio of *relation* increases even at 8 nodes.

Except supporting the conclusions in the previous section, this section also shows that a NOW-based system works very well as a shared-nothing parallel database system even though there are some significant differences between these two systems.

Figure 4.17: Abort ratio with DLKM (lock table in main memory)

4.3.4 Sensitivity to Think Time

Think time indicates how long and how often a transaction is generated and submitted by a terminal to system. Smaller think time increases the system load. The think time in this experiment is set at 10, 20, 30, 60, 120 seconds and distributed uniformly.

Lock Table on Disk

The impact of the think time on the system response time, lock overhead, blocking ratio and abort ratio are discussed in this subsection. The results are given in Figures 4.18, 4.19, 4.20, 4.21, respectively.

For a fixed granularity, the mean response time decreases with increasing think time. This is mainly because larger think times delay the generation of new transactions and thus reduce the average queuing time, which is a part of the response time. The curve of response time becomes flat when the think time reaches 120 sec because the queuing time is very small and can't be reduced much. Note that

Figure 4.18: Response time with DLKM (lock table on disk)

Figure 4.19: Lock overhead with DLKM (lock table on disk)

Figure 4.20: Abort ratio with DLKM (lock table on disk)

increasing think time will not affect the transaction execution time. The response time of *relation* is the lowest, but the difference among *relation*, *page* and *record* is smaller when the think time is larger.

The think time does not affect the execution time of each transaction. The decrease in lock overhead is due to the decrease in abort ratio. Lock contention decreases with larger think times because fewer transactions request locks at the same time. In Figure 4.20, the abort ratio for all granularities reduces with larger think time. The initial increase in abort ratio is due to the same reasons discussed before. When the lock contention is neither very low nor very high, the probability of deadlocks occurring increases.

Lock Table in Main Memory

In this case, the response time also decreases with larger think times as shown in Figure 4.22, except that the value for all granularities is very close.

Figure 4.23 indicates the sensitivity of lock overhead to think time. The crossover

Figure 4.21: Blocking ratio with DLKM (lock table on disk)

Figure 4.22: Response time with DLKM (lock table in main memory)

Figure 4.23: Lock overhead with DLKM (lock table in main memory)

of lock overhead of *record* and *page* is deadlocks having a larger impact on the lock overhead. The lock overhead of *page* is larger than *record* at the beginning because the *page* has the highest abort ratio. When the think time increases, the abort ratio of *page* decreases, which leads to less lock overhead than that of the *record*.

4.3.5 Sensitivity to Data Skew

Data skew is a problem in shared-nothing database systems. To study this issue, the transactions are generated by a normal distribution. The degree of skew is determined by varying a parameter in the distribution, which is the variance. In these experiments, the variance is set at 200, 150, 100 and the degree of data skew is 1, 2, 3 respectively. When the degree of data skew is 0, uniform distribution is used.

Figure 4.24: Response time with DLKM (lock table on disk)

Lock Table on Disk

Figure 4.24 shows the sensitivity of the response time to data skew. For a fixed granularity, with the increasing degree of data skew, the response time increases even though the lock overhead decreases as shown in Figure 4.25. There are two reasons for this behavior. First, when the data accessed by a transaction is not evenly located in all processors, the execution time of the transaction is determined by the slowest task. With higher degree of data skew, more accesses are distributed to one processor, and the job in that processor will slow down the overall response time of the transaction. Assuming that the data skew is very high so that all records to be accessed by a transaction are located in only one processor, the response time will be the same as that without any parallel processing. The lock overhead decreases because transactions tend to request the same lock more frequently, causing the lock I/O overhead to decrease. Second, high degree of data skew causes high lock contention, thus there is more waiting time for each transaction.

With very high data skew, the performance of *page* starts to outperform the

Figure 4.25: Lock overhead with DLKM (lock table on disk)

Figure 4.26: Blocking ratio with DLKM (lock table on disk)

Figure 4.27: Abort ratio with DLKM (lock table on disk)

relation. As shown in the Figure 4.24, when the degree of data skew is 3, the response time of *page* is the best. This is because the *relation* causes more idle time with higher data skew. The response time of *record* is still the worst because its high lock overhead still dominates its performance.

Figures 4.26 and 4.27 show the impact of data skew on blocking ratio and abort ratio. Since the lock contention is extremely high in this case, we can see the abort ratio for both *page* and *relation* decreases while that of *record* increases initially. The abort ratio of *record* decreases finally with much larger data skew. This observation also supports the conclusion we obtained earlier about the generation of deadlocks.

Lock Table in Main Memory

The *record* has a marginally better performance with high data skew as shown in Figure 4.28 (when the degree of data skew is 3). This is mainly due to the fact that the lock overhead of *record* is less in this case.

The lock overhead increases with increasing data skew. This is because, for lock

Figure 4.28: Response time with DLKM (lock table in main memory)

Figure 4.29: Lock overhead with DLKM (lock table in main memory)

table in the main memory, there is no lock I/O overhead. Besides, lock overhead is shared by all processors, thereby with more lock requests in the same node, the lock overhead increases.

4.4 Performance with Centralized Lock Manager

Unlike DLKM, centralized lock manager (CLKM) handles lock requests from different nodes. The advantages of CLKM are that it detects the global deadlock immediately and without any communication. CLKM handles global deadlocks in the same manner as the local deadlocks. One of the disadvantages of CLKM is that there is only one lock table and it may become a hot spot when all transactions submit their requests to the CLKM. Hot spot lock table may cause congestion. The other disadvantage of CLKM is the cost of communication because each lock request has to be submitted by sending a message. With DLKM, lock requests are processed locally.

4.4.1 Sensitivity to System Size

The number of processors in the system is the most important factor for analyzing the difference between a DLKM and a CLKM. Global deadlock occurs when transactions execute at more than one processor. The impact of communication overhead becomes larger with larger number of processors. In this experiment, the parameters are set the same as those in the experiment with DLKM.

Lock Table on Disk

Figures 4.30 and 4.31 show the impact of number of processors on response time and throughput with CLKM respectively. The decrease in response time of *page* and *relation* is due to the same reasons as in DLKM. But the degree of decrease of *page* is reduced from 90.5% with DLKM to 84.4% with CLKM. For *record*, the response time even increases. The *relation* gives the best performance. Similar effects are shown for the throughput.

Figure 4.30: Response time with CLKM (lock table on disk)

Figure 4.31: Throughput with CLKM (lock table on disk)

Figure 4.32: Lock overhead with CLKM (lock table on disk)

Figure 4.33: Abort ratio with CLKM (lock table on disk)

Figure 4.34: Blocking ratio with CLKM (lock table on disk)

The above behavior is due to the fact that CLKM maintains only one lock table. Even though transactions execute in parallel, the lock requests are still processed sequentially. Lock overhead is not shared by processors as in DLKM. As shown in Figure 4.32, the lock overhead for each granularity does not decrease with increasing number of nodes. Furthermore, too many requests going to the single CLKM leads to the CLKM to be a hot spot, especially for fine granularity. This is the reason for the increase in the response time. Therefore, even though processors share the execution time, the response time of *record* actually increases.

Figures 4.33 and 4.34 indicate the change of abort ratio and blocking ratio. Unlike DLKM, the blocking overhead is not shared by processors anymore, and thus it increases with increasing number of processors. As in DLKM, the abort ratio is measured regardless of the number of processors, the abort ratio for CLKM is similar to that of DLKM.

We conclude that coarse granularity such as *relation* is desired in the system with a CLKM, particularly with a large number of processing nodes.

Figure 4.35: Response time with CLKM (lock table in main memory)

Lock Table in Main Memory

The most significant difference between the lock table in memory and on disk is that the response time of *record* decreases substantially. This is because the lock process time is much less than when lock table is on disk. Therefore, the degree of congestion for CLKM is smaller.

Furthermore, when the lock table is in the main memory, there is no significant performance difference among *record*, *page* and *relation* granularities.

4.5 Summary

This chapter presented and discussed the results we obtained from the experiments. Experiments are run by varying different parameters including the *mpl*, transaction size, system size, think time, and degree of data skew. Both distributed lock manger (DLKM) and centralized lock manger (CLKM) are examined. Lock table is assumed in either the main memory or the disks.

Figure 4.36: Throughput with CLKM (lock table in main memory)

Figure 4.37: Lock overhead with CLKM (lock table in main memory)

Figure 4.38: Abort ratio with CLKM (lock table in main memory)

Figure 4.39: Blocking ratio with CLKM (lock table in main memory)

For DLKM and when the lock table is in the main memory, fine granularity such as *record* has marginally better performance only for small transactions. With large transactions, there is no desired granularity. This is mainly because that the lock overhead has less impact when the lock table is maintained in the main memory. For DLKM and when the lock table is on disk, coarse granularity is more efficient. Besides, coarse granularity is easier to implement. For CLKM and when the lock table is in main memory, the performances of all degrees of granularity are close. *Relation* has marginally better performance because of higher degree of congestion due to fine granularity. For CLKM and lock table is on disk, *relation* shows significant performance improvement over *page* and *record*. *Record* is not preferred because it causes severe hot spot problem.

Chapter 5

Conclusion

5.1 Summary and Conclusions

The fundamental issue addressed in this thesis is the desired size of locking granularity in shared-nothing parallel database systems. Due to the trade-off between the lock overhead and concurrency, the choice of granularity has an important impact on the system performance. Both distributed lock manager and centralized lock manager are implemented and studied. Data contention is considered and deadlock is examined. The simulation model in [25] is extended in this study. Instead of modeling the lock conflict artificially, lock manager handles the lock conflicts dynamically. The performance of networks of workstations as a parallel database system is also studied in this thesis.

Chapter 1 presents the issues studied in this thesis and gives a brief introduction of concurrency and locking granularity.

Chapter 2 describes the main parallel database architectures, database partitioning strategies, networks of workstations (NOWs), basic two-phase locking, two-phase commit protocols and deadlocks including both local and global deadlock. Previous work on locking granularity is also discussed.

Chapter 3 describes the database model, workload model and system model. The system design including the structure and behavior of the main components in

the system, and the algorithms and policies employed are discussed in this chapter. This chapter also describes the experimental environment.

Chapter 4 presents and explains the results obtained from the simulation experiments.

The main contribution of the thesis is that it explored the desired granularity in a shared-nothing parallel database system with a realistic database model where the database is a collection of relations. The lock conflict is not modeled artificially. Communication is handled by sending message explicitly.

There are several main conclusions we obtained from the experiments. There is no desired granularity when the lock table is maintained in the main memory. Coarse granularity is sufficient and efficient when the lock table is in disks. NOW-based system works well as a shared-nothing parallel database system even though it has some significant difference with the parallel database system. In system with centralized lock manager, coarse granularity is mandatory.

5.2 Future work

In this study, we assumed that all granularity has the same size. There is no absolute best size of a granule for all applications. Fine granularity is suited for small transactions and when the lock table is in the main memory. Fine granularity is not suitable for applications with large transaction size. Can we let the large size transactions request coarse granularity, while at the same time allowing small transactions to request fine granularity? The question directs the study of locking granularity to a new topic, *hierarchical locking granularity*. Figure 5.1 gives an example of a lock hierarchy. Different transactions can request different locks with size of a *database*, a *relation*, a *page* or a *record*. The concept of hierarchical locking granularity is described in [15]. In the present market, database systems usually applies the same size locking granularity. More attention should be paid to the hierarchical locking granularity.

Figure 5.1: A sample lock hierarchy

References

- [1] R. Agrawal, M. J. Carey, and M. Livny, "Concurrency Control Performance Modeling: Alternative and Implications", *ACM Transactions on Database Systems*, Vol. 12, No. 4, December 1987, pp. 609-654.
- [2] T. E. Anderson, D. E. Culler and D. A. Patterson. "A Case for Networks of Workstations: NOW", *IEEE Micro*, Feb, 1995.
- [3] M. M. Astrahan, "System R: Relational Approach to Database Management", *ACM Transactions on Database System*, Vol. 1, No. 2, June 1976, pp. 97-137.
- [4] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems", *Computing Surveys*, Vol.13, No.2, June 1981, pp. 185-221.
- [5] P. A. Bernstein and E. Newcomer, *Principles of Transaction Processing*, Morgan Kaufmann Publishers, Inc., San Francisco, 1997.
- [6] M. J. Carey and M. Livny, "Parallelism and Concurrency Control Performance in Distributed Database Machines", *Proc. of the 1989 ACM SIGMOD Inter. Conf. on the Management of Data*, Portland, oregon, Vol. 18, No. 2, June 1989, pp. 122-133.
- [7] M. J. Carey and M. Livny, "Distributed Concurrency Control Performance: A Study of Algorithms, Distribution, and Replication", *Proc. 10th VLDB Conference*, Singapore, August 1984, pp. 107-118.

- [8] S. P. Dandamudi, "Using Networks of Workstations for Database Query Operations", *Proc. Int. Conf. Computers and Their Applications*, Tempe, March, 1997.
- [9] S. Dandamudi, S. L. Au, and C. Y. Chow, *A Performance Study of Locking Granularity in Shared-Nothing Parallel Database System*, Technical Report TR-97-4, School of Computer Science, Carleton University, 1997, pp. 1-23.
- [10] B. C. Desai and B. S. Boutros, "Performance of a two-phase commit protocol", *Information and Software Technology*, Vol38, 1996, pp. 581-599.
- [11] D. J. DeWitt, S. Ghandeharizadeh and D. Schneider, "A Performance Analysis of the Gamma Database Machine", *Proc. ACM SIGMOD Conf.*, Chicago, June 1988, pp. 350-360.
- [12] D. DeWitt and J. Gray, "Parallel Database Systems: The Future of High Performance Database Systems", *Communication of the ACM*, Vol.35, No.6, June 1992, pp. 85-98.
- [13] P. A. Franaszek, J. T. Robinson and A. Thomasian, "Concurrency Control for High Contention Environments", *ACM Transactions on Database Systems*, Vol. 17, No. 2, June 1992, pp. 304-345.
- [14] J. N. Gray, R. A. Lorie, and G. R. Putzolu, "Granularity of Locks in a Shared Data Base", *Proceedings of International Conference on VLDB*, Famingham, September, 1975, pp. 428-451.
- [15] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Data Base", *Modeling in Data Base Management Systems*, 1976, pp. 365-394.
- [16] J. N. Gray, *Notes on Database Operating Systems*, in *Operating Systems: An Advanced Course*, Lecture Notes in Computer Science, 1978, pp. 393-481.

- [17] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers, Inc., 1993.
- [18] M. Hsu and A. Chan, "Partitioned Two-Phase Locking", *ACM Transactions on Database Systems*, Vol. 11, No. 4, December, 1986, pp.431-446.
- [19] B. Jenq, B. C. Twichell and T. W. Keller, "Locking performance in a Shared-Nothing Parallel Database Machine", *IEEE Transactions on Knowledge and Data Engineering*, Vol.1, No.4, December 1989, pp. 530-543.
- [20] M. L. Liu, D. Agrawal and A. EL Abbadi, *The Performance of Two-Phase Commit Protocols in the Presence of Site Failures*, Department of Computer Science, University of California.
- [21] C. Mohan, B. Lindsay and R. Obermarck, "Transaction Management in the R Distributed Database Management System", *ACM Transactions on Database Systems*, Vol. 11, No. 4, December 1986, pp. 378-396.
- [22] D. Potier and Ph. Leblanc, "Analysis of Locking Policies in Database Management Systems", *Communications of the ACM*, Vol. 23, No. 10, pp. 584-593.
- [23] *PVM: Parallel Virtual Machine*, The MIT Press, Mass., England, 1994.
- [24] E. Rahm, "Empirical Performance Evaluation of Concurrency and Coherency Control Protocols for Database Sharing Systems", *ACM Transactions on Database Systems*, Vol. 18, No. 2, June 1993, pp. 333-377.
- [25] D. R. Ries and M. Stonebraker, "Effects of Locking Granularity in a Database Management System", *ACM Transactions on Database Systems*, Vol.2, No.3, September 1977, pp. 233-246.
- [26] D. R. Ries and M. Stonebraker, "Locking Granularity Revisited", *ACM Transactions on Database Systems*, Vol.4, No.2, June 1979, pp. 210-227.
- [27] I. K. Ryu and A. Thomasian, "Analysis of Database Performance with Dynamic Locking", *Journal of the ACM*, Vol. 37, No. 3, July 1990, pp. 491-523.

- [28] V. Singhal and A. J. Smith, "Analysis of Locking Behavior in Three Real Database Systems", *The VLDB Journal*, June 1997, pp. 40-52.
- [29] J. F. Spitzer, "Performance Prototyping of Data Management Applications", *Proceedings of ACM 76*, Houston, October, 1976, pp. 287-297
- [30] M. Stonebraker, "The Design and Implementation of INGRES", *ACM Transactions on Database System*, Vol. 1, No. 3, September 1976, pp. 189-222.
- [31] M. Stonebraker, "The Case for Shared Nothing", *Database engineering*, Vol.9, No.1, March 1986, pp. 4-9.
- [32] The Tandem Performance Group, "A Benchmark of Non-Stop SQL on the Debit Credit Transaction", *Proceedings of ACM SIGMOD Conference*, Chicago, June 1988, pp. 337-341.
- [33] A. Thomasian, *Database Concurrency Control: Methods, Performance, and Analysis*, Kluwer Academic Publishers, 1996.
- [34] Y. Xu, *Parallel Hierarchical Database System*, M.C.S. thesis, School of Computer Science, Carleton University, 1993.

Appendix A

This appendix presents the additional data for distributed lock manager (DLKM)

Appendix A. Additional Data for Distributed Lock Manager (DLKM)95

Figure A.1: Throughput with DLKM, (lock table on disk)

Figure A.2: Throughput with DLKM—small transaction, (lock table in main memory)

Figure A.3: Throughput with DLKM—large transaction, (lock table in main memory)

Figure A.4: Lock overhead with DLKM, (lock table in main memory)

Appendix A. Additional Data for Distributed Lock Manager (DLKM)97

Figure A.5: Abort ratio with DLKM, (lock table in main memory)

Figure A.6: Blocking ratio with DLKM, (lock table in main memory)

Appendix A. Additional Data for Distributed Lock Manager (DLKM)98

Figure A.7: Throughput with DLKM, (lock table on disk)

Figure A.8: Throughput with DLKM, (lock table in main memory)

Appendix A. Additional Data for Distributed Lock Manager (DLKM)99

Figure A.9: Lock overhead with DLKM, (lock table in main memory)

Figure A.10: Blocking ratio with DLKM, (lock table in main memory)

Appendix A. Additional Data for Distributed Lock Manager (DLKM)100

Figure A.11: Throughput with DLKM, (lock table on disk)

Figure A.12: Throughput with DLKM, (lock table in main memory)

Figure A.13: Abort ratio with DLKM, (lock table in main memory)

Figure A.14: Blocking ratio with DLKM, (lock table in main memory)

Figure A.15: Throughput with DLKM, (lock table on disk)

Figure A.16: Throughput with DLKM, (lock table in main memory)

Figure A.17: Blocking ratio with DLKM, (lock table in main memory)

Figure A.18: Abort ratio with DLKM, (lock table in main memory)

IMAGE EVALUATION TEST TARGET (QA-3)

APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved