메인메모리 데이터베이스관리시스템에서 고성능 및 고가용성을 위한 트랜잭션 처리 기법

Performance and Availability Enhancement of Transaction Processing for Main Memory DBMS

2005년 2월

仁荷大學校大學院

 컴퓨터·情報工學科

鄭 光 哲
Performance and Availability Enhancement of Transaction Processing for Main Memory DBMS

2005년 2월

指導教授 表 海 英

이 논문을 博士學位 論文으로 提出함

仁荷大學校 大學院
 컴퓨터·情報工學科

鄭 光 哲
Performance and Availability Enhancement of Transaction Processing for Main Memory DBMS

By

Kwang Chul Jung

A DISSERTATION

Submitted to INHA UNIVERSITY in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science and Information Engineering

February 2005
이 논문은 鄭光哲의 博士學位 論文으로 認定함

2005年 2月 日

主番 ______________________

副番 ______________________

委員 ______________________

委員 ______________________

委員 ______________________
Abstract

The recent general database applications on the fields of internet services and mobile communication require the faster response time of transaction as well as the high performance scalability and the high availability. The conventional real-time database systems based on main memory provide the timeliness and predictable execution time of transactions, but they are restricted to be applied to general database applications in the fields because they have strong time-critical constraints that cannot be harmonized with common requirements of the real world. Those systems, exactly, support the predictable response time, but it cannot ensure the scalability and the availability.

This thesis proposes the advanced storage management techniques to acquire the better performance scalability and higher availability on which the real environment has massive transactions as communication areas. To accomplish the purpose the thesis suggests essential parts of transaction processing techniques for fast response time and the scalability as well as a database replication mechanism for high availability. Specifically, there are four techniques as follow.

The concurrency control method of data combines the versioning and the locking mechanisms, which has minimal latching and few locking information. The index method is a B*-tree variant called PML-tree. It includes partial keys, a maximum key and a next link pointer in an index node and provides tree operations considering cache coherence under multiprocessor computers. The checkpoint method is a fuzzy checkpoint that has not any latch on the dirty pages in checkpointing time to enhance the concurrency of ongoing transactions. The database replication method has a lazy and flex scheme without global coordination that allows any node to update any local data and to propagate the updates to the replicas at the destination nodes.
Those methods show better performance than conventional methods and the storage manager adopted of them has better performance scalability and availability than a compared system. The storage manager has been applied into a commercial MMDBMS product and its high performance results are proved by the practical applications of industrial fields.
CONTENTS

LIST OF FIGURE .............................................................................................................. VI
LIST OF TABLE ............................................................................................................. VIII

1  INTRODUCTION ........................................................................................................ 1
   1.1 Research Backgrounds and Motivations .............................................................. 1
   1.2 Research Objectives .............................................................................................. 4
   1.3 Thesis Organization .............................................................................................. 7

2  RELATED WORK ...................................................................................................... 8
   2.1 Characteristics and Comparisons of MMDB Systems .......................................... 8
       2.1.1 General Characteristics of MMDBMS ......................................................... 8
       2.1.2 Comparisons of MMDB Systems ................................................................. 9
   2.2 Conventional Database Concurrency Control ..................................................... 15
       2.3.1 Two-Phase Locking Protocol .................................................................... 15
       2.3.2 Multiversion Concurrency Control ............................................................ 17
   2.3 Index Methods ................................................................................................... 18
       2.3.1 Characteristics of Index in MMDB ............................................................... 18
       2.3.2 Problems of T-tree .................................................................................... 19
       2.3.3 Cache Conscious Index ............................................................................. 22
   2.4 Logging and Recovery ......................................................................................... 23
2.5 Replication Protocols................................................................................................... 26

3 TRANSACTION PROCESSING TECHNIQUES FOR PERFORMANCE........... 30

3.1 MVSL: Multiversion with Simple Locking Concurrency Control ......................... 30

  3.1.1 MVSL Protocol..................................................................................................... 30

  3.1.2 Version Management............................................................................................ 34

  3.1.3 Locking Management........................................................................................... 38

  3.1.4 Concurrency Control Algorithms ......................................................................... 43

3.2 PML-tree: Cache Conscious Tree Index.................................................................... 47

  3.2.1 Index Structure ..................................................................................................... 47

  3.2.2 Operations on a PML-tree .................................................................................... 49

  3.2.3 Time and Space Analysis ..................................................................................... 54

4 TRANSACTION PROCESSING TECHNIQUES FOR STABILITY AND
AVAILABILITY........................................................................................................... 57

4.1 Multilevel Logging and Recovery ........................................................................... 57

  4.1.1 Multilevel Logging............................................................................................... 58

  4.1.2 Latch-free Fuzzy Checkpoint ............................................................................... 60

  4.1.3 Restart Recovery................................................................................................... 63

4.2 Lazy-flex Replication................................................................................................. 66

  4.2.1 Replication Model ................................................................................................ 66

  4.2.2 Replication Manager Architecture ....................................................................... 70

  4.2.3 Various Synchronization Protocols ....................................................................... 71
LIST OF FIGURE

Figure 1.1 Database Access Pattern of HLR on Real Environment ........................................ 2
Figure 1.2 Running Environment of Access Point on HPi.................................................... 3
Figure 1.3 Application Environments of MMDBMS on Communication Area .................. 6
Figure 2.1 T-tree Index Structure Modification with INV .................................................. 21
Figure 2.2 CPU-memory Performance Imbalance .................................................................. 22
Figure 2.3 Comparisons of Eager and Lazy Scheme ............................................................ 27
Figure 2.4 Comparisons of Primary-copy and Update-Everywhere Scheme .................... 28
Figure 3.1 SCN: Version Number Structure ........................................................................ 35
Figure 3.2 An Example to Find Version ............................................................................. 36
Figure 3.3 Version List Configurations ............................................................................. 38
Figure 3.4 Control Structure of Table Lock Management .................................................. 41
Figure 3.5 Internal Node Structure ..................................................................................... 49
Figure 3.6 Typical Node Split in the Traditional B’-tree ..................................................... 53
Figure 3.7 Left-most Nodes Split in the PML-tree .............................................................. 54
Figure 4.1 Full Replication Topology in Lazy-flex Scheme .................................................. 67
Figure 4.2 Replications Models in Lazy-flex Scheme ......................................................... 68
Figure 4.3 Replication Manager Architecture ..................................................................... 71
Figure 4.4 Coordination Flow between Sender and Receiver .......................................... 72
Figure 4.5 Synchronization Flow of Update Transaction .................................................... 75
Figure 5.1 Storage Manager of ALTIBASE ................................................................. 82
Figure 5.2 Performance Evaluation Model ............................................................ 86
Figure 5.3 Comparisons of Concurrency Control Methods ................................. 87
Figure 5.4 Concurrency of Various Transactions Combination on MVSL .......... 87
Figure 5.5 Effect of Transaction Distribution (read 70%, write 30%) ................. 88
Figure 5.6 TPS for Increasing the Number of Indexes ......................................... 89
Figure 5.7 TPS of PML-tree and T-tree INV for Concurrency ......................... 90
Figure 5.8 Performance of each Logging Level ................................................... 92
Figure 5.9 Transaction Durability vs. Scalability ............................................... 92
Figure 5.10 Effect of Checkpoint on Transaction Processing ............................ 93
Figure 5.11 Recovery Times vs. Log Size ............................................................ 94
Figure 5.12 Throughputs for Various Replication Models ................................. 96
Figure 5.13 Scalability of the Storage Management System ............................ 97
Figure 5.14 TPC-H Benchmark Results ............................................................. 98
Figure 5.15 Wisconsin Benchmark Results ......................................................... 99
Figure 5.16 Effects of Data Types on Performance ............................................. 99
LIST OF TABLE

Table 2.1 Summary of MMDB Systems ................................................................. 14
Table 2.2 Classification of Replication Management Strategies ......................... 29
Table 3.1 Lock Compatible Table ......................................................................... 42
Table 3.2 Lock Conversion Table .......................................................................... 42
Table 3.3 Values and Parameters for Time and Space Analysis ......................... 55
Table 4.1 Savepoint in proposed method ............................................................... 60
Table 4.2 Node Types in Lazy-flex Replication ................................................... 68
Table 4.3 Inconsistent States in Deferred Replication Scheme ............................ 79
Table 4.4 Data Conflict Resolution Policies ......................................................... 80
Table 5.1 Environment for Experiments ............................................................... 85
Table 5.2 TPS of Single User Environment ........................................................... 97
1 Introduction

1.1 Research Backgrounds and Motivations

The recent database applications require the short and predictable transaction time. The characteristics of real-time database are also expected in the general database applications such as call routing and switching of telecommunications and mobile applications [JLB03]. For example, in the case of practical system of internet portal site, there are 1,000 or more the user authentication transactions that should be processed at a same time. The home location register (HLR) system on mobile communication has to process the 2,500 calls per a second in mobile communication environments. In a view of DBMS perspective, the capacity of DBMS performance is required to process more than 15,000 database operations per a second.

HLR is the system equipment that provides the service for location information management, terminal authentication and connection management in mobile communication environment. In order to analyze the actual database operations (read and write) for HLR, the system was monitored and traced with its SQL statements that were executed in all days. The access pattern of database is classified into read operations which are select SQL statements and write operations which are other SQL statements. The access pattern in HLR system is described in Figure 1.1. The entire operations consist of read operation as 70% ratio and write operation as 30% ratio. The database operation processing rate per a second is placed between 3,000 and 15,000. HLR system requires the real time response in the hot spot time, and the performance and concurrency over the read operation should be enhanced in a view of database perspective.
The recent database applications such as high portable internet (HPi) and CDMA are required in the new computing paradigm (e.g., diskless system) beyond the conventional disk-memory-CPU environment. In this environment, the access point (AP) system loads the database from the remote database server. If the access point system is powered off, all the database should be destroyed in the access point system and whole database should be re-loaded from remote server when the system is turned-on. The database durability is not ensured but the consistency of database on running state should be managed. On the other hand, the recent database state of AP system should be reflected into the original remote database server periodically in order to keep up with the consistent database state at the fixed time.
Online stock trading systems, real-time billing systems, and soft switch systems are representative fault-tolerant example systems [JLB04]. These DBMS application systems also require the fast response time as well as the system availability. As a real system, the real-time billing system of SKT that is the mobile communication service company has the fully replicated database systems of 4-ways for availability as well as load balancing. Each database system could restore the amount of billing data for three months in memory, which database size is uppermost 40 GB, and process 3,500 calls per second and three hundred millions calls per day in maximum. The general-purpose disk resident database systems provide the database clustering for the system availability. However, the disk resident database systems failed to meet the needs of applications requiring short and predictable execution time, since they are optimized for the characteristics of disk storage environment. Real-time database systems provide the timeliness and predictable execution time of transactions, but they are restricted to be applied to general database applications because they have strong time-critical constraints that cannot be harmonized with common requirements of the real world. Also conventional real-time database systems
don’t support the fault-tolerant database system because they use the dedicated main memory on the hardware platforms for storing the database. Hence current DBMS applications need both of the fast response requirement from real-time DBMS and the high availability from fault-tolerant disk-based DBMS.

As described above, the requirements of database applications on internet services and mobile communications can be summarized as the followings.

• Concurrent users can be extremely increased at a certain time and each transaction requires the fast response time. In other words, 1,000 concurrent users and 15,000 or more than database operations are required.

• There are 70% read operations and 30% write operations. The fast response time and the higher concurrency are required especially for the read operations.

• The data conflict is rarely occurred and even though there is a data conflict, sometimes the read of old-date data can be allowed.

• There is a trade-off relationship between the transaction durability and performance. The transaction durability can be sacrificed if the fast response time is more important feature of database application. Also, fast recovery is required when the database system failure occurs.

• The high availability of DBMS can be achieved by the fault-tolerant facility of DBMS itself rather than the one of computer systems. Also, the various configurations of replicated database servers are required, e.g., active-standby, active-active, and active-multi standby.

1.2 Research Objectives
Most disk-based database systems are designed to maximize the overall transaction throughput, rather than provide fast and predictable response time per individual request of transactions. Traditional disk-based database systems, therefore, are incapable of achieving the latter goal due to the latency of accessing data that is disk-resident. An attractive approach to providing predictable response per each request is to load the whole data into main memory [GMS93]. It can be suggested by the increasing availability of large and relatively cheap memory. Most database servers with main memories of several gigabytes or more are already available and common. The disk-resident database system (DRDB) with a very large buffer, however, is basically different from pure main memory database systems (MMDB). The key difference is that there is no need to interact with a buffer manager for fetching or flushing buffer pages. The performance of MMDB can be improved further by dispensing with buffer manager, changing the storage hierarchy into the flat structure [GMS93, BPR’96]. In a MMDB, the whole database can be directly mapped into the virtual memory address space. Also, MMDB has almost features of the conventional elements of DRDB, such as data organization, concurrency control, transaction and index management. The implementation techniques and storage management method of MMDB have been researched because those techniques of DRDB cannot be directly adopted into MMDB [BLR’97]. The methods in those researches have to be improved to obtain the high performance and availability. It is difficult for those techniques to be implemented into the commercial MMDBMS because of the lack of stability and the practice.

Therefore, there are needs of design and implementation of new main memory database management system that is adequate to the application to require the high performance and availability under the real environment of mobile communication and internet services as illustrated Figure 1.3. This thesis mainly proposes the essential techniques of transaction processing such as the concurrency control method, the index techniques, the logging and recovery, and the replication mechanisms. It also presents how the proposed
methods are improved upon conventional methods and how those techniques are implemented into the practical commercial system.

![Diagram of Application Environments of MMDBMS on Communication Area]

Figure 1.3 Application Environments of MMDBMS on Communication Area

The objectives of the thesis are as following: First, to implement the advanced storage manager in main memory DBMS that is adopted on database services of real communication area, draws the requirements of the database services from analysis of required transaction pattern, performance, stability, and availability on those database services. Secondly, to provide the transaction processing techniques with performance scalability and availability, the thesis suggests the more efficient concurrency control method, index management, logging and checkpoint management, and replication mechanisms in a storage management system. Thirdly, the performance of proposed methods is experimented in the real system environment. The performance evaluations of
the proposed methods are exploited with the various experimental parameters. Finally, the proposed methods are implemented into the actual MMDBMS and showed the efficiency and improvements of the methods by using the standard test suits for DBMS.

1.3 Thesis Organization

The thesis is organized as follows. The next section 2 explains the related works for conventional MMDB systems and traditional techniques for implementing the main memory database systems. Section 3 proposes transaction processing techniques to enhance the response time and performance scalability such as multiversion with simple locking concurrency control method and cache conscious tree index method. Section 4 explains the advanced methods for transaction processing for stability and availability of database such as multilevel loggings, an efficient checkpoint method, and a replication method. In section 5, the transaction performance of proposed methods is evaluated under various circumstances and the conclusion is described in section 6.
2 Related Work

2.1 Characteristics and Comparisons of MMDB Systems

2.1.1 General Characteristics of MMDBMS

In a main memory database system (MMDB) data resides permanently in main physical memory; in a conventional database system (DRDB) it is disk-resident. In a DRDB, disk data may be cached into memory for access; in a MMDB the memory resident data may have a backup copy on disk. So in both cases, a given object can have copies both in memory and on disk. The key difference is that in MMDB the primary copy lives permanently in memory, and this has important implications as to how it is structured and accessed [GMS93].

A computer’s main memory clearly has different properties from that of magnetic disks, and these differences have profound implications on the design and performance of the database system. Although these differences are well known, it is worthwhile reviewing them briefly:

- The access time for main memory is orders of magnitude less than for disk storage.

- Main memory is normally volatile, while disk storage is not. However, it is possible at some cost to construct nonvolatile main memory.

- Disks have a high, fixed cost per access that does not depend on the amount of data that is retrieved during the access. For this reason, disks are block-oriented storage devices. Main memory is not block oriented.
The layout of data on a disk is much more critical than the layout of data in main memory, since sequential access to a disk is faster than random access. Sequential access is not as important in main memories.

Main memory is normally directly accessible by the processor(s), while disks are not. This may make data in main memory more vulnerable than disk-resident data to software errors.

Main memory databases feature all the conventional elements that one would expect in a database system, namely: data organization, access methods, concurrency and deadlock management, query processing and optimization, commit protocols and recovery. In standard database systems, most of the above operations and functionalities are designed around the movement of data blocks/pages in the memory hierarchy. In an MMDB, the fundamental difference is that its components are designed to take advantage of the fact that data do not need to be transferred from disks [GMS93].

2.1.2 Comparisons of MMDB Systems

Several database management systems for memory resident data have been proposed or implemented. The efforts range from pencil-and-paper designs (MM-DBMS) to prototype or testbed implementations (System M, Dali, Mr.RT) to commercial systems (TimesTen) [GMS93]. The descriptions of the systems are necessarily brief. Furthermore, the focus of the descriptions on how these systems address the issues raised by memory resident data, which the thesis will propose the key technologies of transaction processing in a storage manager.

MM-DBMS
The MM-DBMS system was designed at the University of Wisconsin [LC86, LC87]. This system implements a relational data model and makes extensive use of pointers for data representation and access methods. Index structures point directly to the indexed tuples, and do not store data values. A variant linear hashing is used to index unordered data, and T-trees are used to access ordered data.

For recovery purposes, memory is divided into large, self-contained blocks. These blocks are the units of transfer to and from the backup disk resident database copy. Commit processing is performed with the aid of some stable memory for log records and with a separate recovery processor. The recovery processor groups log records according to which blocks they references so that blocks can be recovered independently after a failure. Blocks are checkpointed by recovery processor when they have received a sufficient number of updates. A lock is set during the checkpoint operation to ensure that each block is in a transaction consistent state on the disk. After a failure, blocks are brought back into memory on demand and brought back up to data using log record groups. MM-DBMS uses two-phase locking for concurrency control. Large lock granules, relations, are used.

System M

System M is transaction processing testbed system developed at Princeton for main memory databases [SGM90]. This system is designed for a transactional workload rather than adhoc database queries. It supports a simple record-oriented data model.

System M is implemented as a collection of cooperating servers (threads) on the Mach operating system. Message servers accept transaction requests and return results to clients. Transaction servers execute requested transactions, modifying the database and generating log data. Log servers move in-memory log data to disk, and checkpoint servers keep the disk resident backup database up to date.
System M is capable of processing transactions concurrently. However, it attempts to keep the number of active transactions small. Two-phase locking is used for concurrency control. Both pre-commit and group-commit are implemented for efficient log processing.

As in MM-DBMS, the primary database copy is divided into self-contained fixed-size segments, which are the units of transfer to and from the backup disks. Record index structures reside outside of the segments since they are not included in the backup database (nor are changes to indices logged). Indices are recreated from scratch after a failure, once the database has been restored from the backup copy and the log.

Since the focus of System M is empirical comparison of recovery techniques, a variety of checkpointing and logging techniques are implemented. System M can perform both fuzzy and consistent checkpoints and both physical and logical logging. The physical organization of the backup database copy can also be controlled.

**Dali**

Dali is main memory storage manager designed and implemented at AT&T Bell Labs [BLR’97]. Dali system consists of a set of database files along with one or more server processes. A Dali system runs in a shared virtual memory that guarantees sequential consistency for reads and writes. Server as well as user processes with access to the shared virtual memory can map data into their virtual memory address space; other processes can access data only via processes that have access to the shared virtual memory.

Dali provides the multiversion concurrency control with two-phase locking protocol. This allows read-only transactions to execute without obtaining data item locks and update transactions to execute with obtaining data item locks to for high concurrency. As an access method, Dali provides the T-tree that is with logical and physical versioning. When
the nodes of T-tree are updated, Dali uses the index node versioning for that the read operations can access the nodes without waiting the completion of the index structure modification.

Dali recovery is the integration of ping-pong checkpointing with multi-level recovery. The ping-pong checkpointing uses the two physical copies of the checkpoint image to overcome that fuzzy checkpointing can lead to potential violations of the write-ahead logging rule. The recovery manager of Dali provides support for physical logging as well as logical logging. Recovery must be performed using logical logging for high concurrency structures such as storage allocation tables and indices.

Mr.RT

Mr.RT is a storage management system for memory resident real-time database designed and developed at Electronics and Telecommunications Research Institute (ETRI) [PLK95]. Mr.RT is consists of several managers. The transaction scheduler determines the execution order of transactions. The storage manager manages the primary database in main memory and provides routines for creating, deleting, and accessing records, indices, etc. Index manager provides both hash index and tree index. Hash index is an extensible chained bucket hash (ECBH) to index unordered data, which number of buckets is increasing when the indexed data items are enlarged. As an access method, Mr.RT uses the T-tree index for range or exact match access.

The concurrency control protocol of Mr.RT is a form of two-phase locking with relation lock granule. By using high priority protocol, the system resolves a conflict in favor of the transaction with the higher priority. It aborts the low-priority lock holder and lets the high-priority lock requestor proceed.
The log of Mr.RT consists of two parts, a memory-resident buffer and a stable log volume that resides on disk. The memory resident buffer holds the log tail, the most recently created part of the log. Stable memory can be used as a write log buffer between memory resident buffer and disk. With stable memory, transactions can commit without disk I/O.

The checkpoint manager is responsible for migrating changes in the primary database to the backup. Checkpointing should interfere as little as possible with transaction processing. In Mr.RT, the database is divided into segments. Different segments in the database can have different checkpointing schemes. For example, segments holding database structures such as indices that are re-computable during recovery, does not need checkpointing at all. Mr.RT supports the fuzzy checkpointing as its default.

**TimesTen**

TimesTen is a commercial product for main memory database system produced at TimesTen, Inc. [TT]. TimesTen/DataServer provides real-time capture, management, and distribution of data for applications requiring exceptional performance, and supports the failover, recovery, and evolution of database with minimal downtime.

For concurrency control, TimesTen provides the two-phase locking protocol with various lock items, database, relation, and tuple. Tuple-level locking provides the multi-user concurrency, and is the default. And the isolation level of transaction on TimesTen has two isolation levels: serializable and read-committed. As an access method, it provides a T-tree index with latching and lock coupling for concurrency control of among the index operations. For the recovery, also, TimesTen uses the various logging options, the fuzzy checkpointing, and the recovery mechanism based on ARIES [MHL’92].
Table 2.1 Summary of MMDB Systems

<table>
<thead>
<tr>
<th>Method System</th>
<th>Concurrency</th>
<th>Commit processing</th>
<th>Access methods</th>
<th>Recovery</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM-DBMS</td>
<td>Two-phase locking of relations</td>
<td>Stable log tail by segment</td>
<td>Hashing, T-tree, Pointers to values</td>
<td>Segments recovered on demand, recovery processor</td>
</tr>
<tr>
<td>System M</td>
<td>Two-phase locking, minimize concurrency</td>
<td>Several alternatives</td>
<td>Hashing, T-tree</td>
<td>Various checkpointing, logging options</td>
</tr>
<tr>
<td>Dali</td>
<td>Multiversion</td>
<td>Pre-commit</td>
<td>T-tree</td>
<td>Ping-pong fuzzy checkpoint</td>
</tr>
<tr>
<td>Mr.RT</td>
<td>Two-phase locking of relations</td>
<td>Stable log tail or pre-commit</td>
<td>Hashing(ECB H), T-tree</td>
<td>Fuzzy checkpoint</td>
</tr>
<tr>
<td>TimesTen</td>
<td>Two-phase locking of tuples, relations and database</td>
<td>Pre-commit depend on logging option</td>
<td>T-tree</td>
<td>Fuzzy checkpoint, various logging option</td>
</tr>
<tr>
<td>ALTIBASE</td>
<td>Multiversion with simple locking</td>
<td>Several alternatives depend on logging level</td>
<td>Modified B’-tree</td>
<td>Latch free fuzzy checkpoint and Multilevel logging</td>
</tr>
</tbody>
</table>

Table 2.1 shows the comparisons of transaction processing techniques that are adopted in the conventional MMDB systems. Those systems support the management techniques of database resided in main memory and transaction processing techniques in single processor environment. Thus those systems have a limitation to fully support the requirements of database application on communication areas that are described in chapter 1. Especially they cannot support the high scalability for thousand of concurrent users and the high availability of database. The ALTIBASE [JLB03, JLB04] is a commercial MMDBMS that is reflected the transaction processing techniques which are proposed in this thesis.
2.2 Conventional Database Concurrency Control

To enforce serializability of transactions, many protocols have been devised. Among the many techniques, two-phase locking [MHL’92] and versioning [AK91, BC92] are widely cited and used. With respect to MMDBMS, they both have drawbacks. This section discusses them and their associated problems. The modified multiversion concurrency control method proposed by this thesis is a combination of these two basic methods.

2.3.1 Two-Phase Locking Protocol

A transaction is said to satisfy the two-phase locking (2PL) protocol [MHL’92] if all of its locking operations precede the first unlock operation. A transaction satisfying this protocol consists of 2 phases - the growing phase during which new locks are acquired but no locks can be released and the shrinking phase during which locks are released but no new lock can be acquired.

One problem with 2PL is deadlock. Deadlock can occur when two or more transactions are involved in a circular waiting, each waiting for the other to release some lock but itself also holding a lock that is need by other transaction involved in the waiting. Predicting the cost of detecting and recovering from deadlock is often too pessimistic. Hence, in most real-time systems deadlock prevention mechanisms are used instead. This means that if 2PL is to be used in the transaction scheduling, one must use appropriate methods to prevent deadlock, e.g. by requiring that data items be requested in some prevented order. In this case, one trades flexibility and efficiency for predictability.

Another problem with 2PL occurs the system includes relatively long transactions. 2PL can cause excessive blocking of other short transactions, which could severely affects performance scalability of the system.
In here, when the locking method is applied to main memory database system, some problems are discussed precisely. Because access to main memory is so much faster than disk access, a database can expect transactions to complete more quickly in a main memory system. In the past systems that use lock-based concurrency controls, this means that locks will not be held as long, and suggests that lock contention may not be as important as it is when the data is disk-resident. Systems that choose small locking granules (fields or records) do so to reduce contention. If contention is already low because data is memory-resident, the principal advantage of small lock granules is effectively removed. For this reason, it has been suggested that very large lock granules (e.g., relations) are most appropriate for memory-resident data [LC87]. In the extreme, the lock granule could be chosen to be the entire database [GMS87, LN88].

The big lock granule, however, reduces the concurrency under the recent high fast multiprocessor computers. Serial transaction processing is highly desirable, since the costs of concurrency control are almost completely eliminated. Furthermore, the number of CPU cache flushes is greatly reduced. Each time a transaction is suspended waiting for a lock, a new transaction is run and the contents of the CPU cache must change. With serial execution, only one flush needs to occur per transaction. In high performance computers, where cache flushes are equivalent to thousands of instructions, the gains can be very significant. However, serial transactions are probably not practical when long transactions (e.g., conversational transactions) are present. For fairness, there should be some way to run short transactions concurrently with long-lived ones. Furthermore, multiprocessor systems may require some form of concurrency control even if all transactions are short.

The actual implementation of the locking mechanism must have a big overhead to manage locking information. In case of main memory database systems, especially, the locking mechanism has some disadvantages such that there is a lot of information for locking
objects (e.g. records and tables). This problem is more serious when a number of concurrent transactions is larger or when the space for locking information management is restricted. In a conventional system, locks are implemented via a hash table that contains entries for the objects currently locked. If there are many locked objects, the hashing on the hash table is very slow and managing the hash table can be complex.

2.3.2 Multiversion Concurrency Control

Many applications in telecommunications require very fast and predictable response times for transactions and, in particular, for read-only transactions. Since disk I/O in an MMDB is only needed for persistence of the log, no disk activity is required on behalf of read-only transactions. As a result, response times for read-only transactions are more predictable, making MMDBs highly suitable for a large class of real-time applications. However, a read-only transaction may still have to wait on locks held by an update transaction, which may in turn be waiting on a different transaction, or on disk writes to the log. These waits become a serious source of unpredictability for response times.

Multiversion concurrency control methods [AK91, BC92, RSB’97] prevent update transactions from conflicting with read-only transactions by providing the latter with a consistent but somewhat out-of-date view of the database. In order to provide this view, multiple versions of recently updated data items are retained. Early multiversion schemes used timestamps to serialize readers as well as writers, but other schemes [BC92, MHL’92] use timestamps to serialize read-only transactions with respect to updaters, allowing them to use old versions without locking, while requiring updaters to perform locking to serialize themselves with respect to other updaters.

However, none of the above techniques guarantees complete isolation of read-only
transactions from update transactions in a system, since the access path to the data could be modified by update transactions. Thus, read-only transactions must obtain latches to ensure that they read physically consistent data. To solve this problem, Dali system [RSB’97] assigns version number (logical timestamp) into all versions. Read-only transaction in the system checks the version number and read the version owns itself. The read-only transaction, at that time, dose not acquires any latches at all. Dali system has high concurrency of read-only transactions, but it can not improve the concurrency of update transactions because it has been used the conventional locking mechanism to solve the contention of among update transactions. This thesis, thus, suggests a new concurrency control method that is efficient for which the read and update transactions have been occurred excessively.

2.3 Index Methods

2.3.1 Characteristics of Index in MMDB

In a main memory database, index structures like B-trees, which are designed for block-oriented storage, lose much of their appeal. A wide variety of index structures have been proposed and evaluated for main-memory databases [Dea84, LC86, WK90]. These include various forms of hashing and of trees. Hashing provides fast lookup and update, but may not be as space-efficient as a tree, and does not support range queries well. Trees such as the T-tree have been designed explicitly for memory-resident databases [LC86]. Main-memory trees need not have the short, bushy structure of a B-tree, since traversing deeper trees is much faster in main memory than on a disk.

One observation common to all main memory access methods is that the data values on which the index is built need not be stored in the index itself, as is done in B-trees.
Because random access is fast in main memory, pointers can be followed quickly. Therefore, index structures can store pointers to the indexed data, rather than the data itself. This eliminates the problem of storing variable length fields in an index and saves space as long as the pointers are smaller than the data they point to.

The use of pointers suggests perhaps the simplest way to provide an index, which is simply to invert the relation on the indexed field [AHK85, BHT87, WK90]. In a main memory database, the inverted “relation” can simply be a list of tuple pointers in sorted order. Such indices are very space efficient and reasonably fast for range and exact-match queries, although updates are relatively slow.

Main memory databases can also take advantage of efficient pointer-following for data representation. Relational tuples can be represented as a set of pointers to data values [PTV90, WK90]. The use of pointers is space efficient when large values appear multiple times in the database, since the actual value need only be stored once. Pointers also simplify the handling of variable length fields since variable length data can be represented using pointers into a heap [LC87, SGM90].

The recent computers have CPU caches (L1 and L2), however, the pointer operations to find indexed data have been leaded to cache misses frequently. Cache miss is very seriously to degrade the performance of the index. So, the use of pointers to the indexed data is not that is good every time.

2.3.2 Problems of T-tree

The T-tree index structure was adopted by several MMDBMS such as the Starburst system [LC92] and Dali system [RSB’97], because of better performance than the conventional B-tree. In those systems, latching and locking for concurrent access to the
index structure are major factors that dominate the cost of database access since the I/O bottleneck of paging data into and out of the main memory are removed. Thus the performance of concurrent access T-tree over the B-tree is the important consideration point for the storage management to adopt an index structure. As pointed by Lu et al [LNT00], the T-tree does not provide a better performance than the B-tree when the concurrent access from multiple users is allowed because of the high cost of locking required to enforce concurrency control.

Hence, the T-tree with index node versioning (INV) method [RSB’97] proposed that all the read operations can be executed without any latch and locking operation, since insert, delete, and update operations over the index structure are executed on the new version of T-node under the multiversion technique. For each operation that modifies an index structure, a new version of T-node should be created and the operation should be executed on the new version of T-node. Afterwards, the previous node pointer is changed to point a new version of T-node or subtree. The read operation thus can be performed without any latch and lock operation in a whole index structure. The T-tree of [JLB03] can be traversed more efficiently through the physical versioning of a T-node upon the change operation on the T-tree.

Let the component is the set of index node that is affected by index structure modification operation and $N$ is the root node of component. As depicted in Figure 2.1, the index node versioning operation is performed as following steps [JLB03].

1. The component is copied and it is called as $N'$.

2. The update operation is performed upon the copied component $N'$. The existing node can be destroyed and the new node can be added into $N'$. The original index tree however is not affected by the update operation.
3. The previous pointer $N$ is changed as the $N'$. Other transactions are not affected by the operation since it is performed atomically.

Transactions see the different version of index node in the symmetric multiple processor with two or more CPUs. Each CPU has its own cache and there are different versions of index nodes in those caches. So the index node versioning could be assured that the transaction execution is correct unless the old version index node $N$ is not eliminated until the transaction will not be referenced it.

The major problem, however, of the T-trees with INV [RSB’97, JLB03] is the high cost of creating and managing versions during the tree structure modifications. The indexes performance degrades sharply with the increasing the update ratio. The scalability of update performance is also very poor, even on the four processors platforms where reported experiment was conducted.

Figure 2.1 T-tree Index Structure Modification with INV
2.3.3 Cache Conscious Index

As random access memory gets cheaper, it becomes increasingly affordable to build computers with large main memories. The Asilomar Report [BBC’98] says: “Within ten years, it will be common to have a terabyte of main memory serving as a buffer pool for a hundred-terabyte database. All but the largest database tables will be resident in main memory.” But main memory data processing is not as simple as increasing the buffer pool size. An important issue is cache behavior. The traditional assumption that memory references have uniform cost is no longer valid given the current speed gap between cache access and main memory access. A significant portion of execution time in commercial main memory database management systems is spent on second level data cache misses and first level instruction cache misses [ADW99]. Further more, CPU speeds have been increasing at a much faster rate (60% per year) than memory speeds (10% per year) as shown in Figure 2.2. So, improving cache behavior is going to be an imperative task in main memory data processing.

![Figure 2.2 CPU-memory Performance Imbalance](image)

Index structures are important even in main memory database systems. Although there are
no disk accesses, indexes can be used overall computation time without using too much extra space. Index structures are useful single value selection, range queries and indexed nested loop joins. With a large amount of RAM, most of the indexes can be memory resident. [RR00] studied the performance of main memory index structures and found that B’-trees are more cache conscious than binary search trees and T-trees [LC86].

The so-called cache conscious index structures such as the CSB’-tree [RR00] reduce the cache misses and thereby improve the search performance. The CSB’-tree keeps only one child pointer of the B’-tree per node, almost doubling the fanout of the tree. While all of cache conscious indexes effectively improve the search performance by increasing the index node fanout and reducing the capacity cache misses, they were studied without much consideration of concurrency control among the tree operations, which is crucial for running the real-world main memory database applications involving index updates and taking advantages of the multiprocessor platforms for scaling up the performance of such applications. So, this thesis studies a new index method considering cache conscious index structure and index operations for high performance and scalability.

2.4 Logging and Recovery

To protect against media failures, it is necessary to have a backup copy and to keep a log of transaction activity. Since memory is usually volatile, this log must reside in stable storage (e.g., redundant disks). Before a transaction can commit, its activity records must be written to the log [GR92]. The need for a stable log threatens to undermine the performance advantages that can be achieved with memory-resident data. Logging can impact response time, since each transaction must wait for at least one stable write before committing. Logging can also affect throughput if the log becomes a bottleneck. Although these problems also exist when data is disk-resident, they are more severe in main memory systems because the logging represents the only disk operation each
transaction will require.

Several solutions have been suggested for this problem. First, a small amount of stable main memory can be used to hold a portion of the log [Dea84, Hag86, Eic87a, Eic87b, GMS87, LC87]. A transaction is committed by writing its log information into the stable memory, a relatively fast operation. A special process or processor is then responsible for copying data from the stable memory to the log disks. Although stable memory will not alleviate a log bottleneck, it can eliminate the response time problem, since transactions need never wait for disk operations. Studies have suggested that only a small amount (e.g., fewer than one hundred log pages [CS89]) of stable memory is needed to hold the log tail, even in high-performance systems.

In case stable memory is not available for the log tail, transactions can be pre-committed [Dea84, GK85]. Pre-committing is accomplished by releasing a transaction’s locks as soon as its log record is placed in the log, without waiting for the information to be propagated to the disk. The sequential nature of the log ensures that transactions cannot commit before others on which they depend. Although pre-committing a transaction does not reduce its response time, it may reduce the blocking delays of other concurrent transactions.

There is another technique called group commits in case stable memory is not available for the log tail. Group commits can be used to relieve a log bottleneck [Dea84, GK85]. Under group commit, a transaction’s log record need not be sent to the log disk as soon as it commits. Instead, the records of several transactions are allowed to accumulate in memory. When enough have accumulated (e.g., when a page is full), all are flushed to the log disk in a single disk operation. Group commit reduces the total number of operations performed by the log disks since a single operation commits multiple transactions. Both techniques, pre-committed and group commits, cannot ensure the data consistency if the
system failure occurs when the transaction’s logs was not yet flushed into disk.

Backups of memory-resident databases must be maintained on disk or other stable storage to insure against loss of the volatile data. Recovery has several components, the first being the procedure used during normal database operation to keep the backup up-to-date, and the second being the procedure used to recover from a failure. The previous researches have already discussed commit processing, which is used to make sure that the results of all committed transactions are stable. Most systems that use a log for commit processing also perform backups or checkpoints to limit the amount of log data that must be processed to recover from a failure [Dea84, Hag86, Eic87a, Eic87b, LC87, LN88, SGM89, SGM90]. Checkpointing brings the disk resident copy of the database more up-to-date, thereby eliminating the need for the least-recent log entries.

In a memory-resident database system, checkpointing and failure recovery are the only reasons to access the disk-resident copy of the database. Application transactions never require access to the disk resident data. Therefore, disk access in a memory-resident system can be tailored to suit the needs of the checkpointer alone. One observation is that disk I/O should be performed using a very large block size. Large blocks are more efficiently written, and though they take longer, only the checkpointer, and not the application transactions, awaits the completion of those writes.

Checkpointing should interfere as little as possible with transaction processing. Transaction-consistent or action-consistent checkpoints require some synchronization (e.g., locking) with transactions. An alternative known as fuzzy dumping requires no synchronization. However, consistent checkpoints may simplify logging, since logical operations can be logged.

After a failure, a memory-resident database manager must restore its data from the
disk-resident backup and then bring it up to date using the log. If the database is large, simply transferring the data from the disks may take a long time. One possible solution to this problem is to load blocks of the database “on demand” until all of the data has been loaded [LC87, GE91]. However, it is not clear how much of an improvement this will provide in a high-performance system, which must handle the demands of thousands of transactions in the seconds after the database has recovered.

Another possible solution to the database restoration problem is to use disk striping or disk arrays [Kim86, SGM86, Pea88]. Here the database is spread across multiple disks, and it is read in parallel. For this to be effective there must be independent paths from the disks to memory.

2.5 Replication Protocols

Replication has been studied in many areas, especially in distributed systems (mainly for fault tolerance purposes) and in databases (mainly for performance reasons). In these two fields, the techniques and mechanisms used are similar, and yet, comparing the protocols developed in the two communities is a frustrating exercise that many researchers have unsuccessfully attempted. Due to the many subtleties involved, mechanisms that are conceptually identical, end up being very different in practice. As a result, it is very difficult to take results from one area and apply them in the other. In the last few years, as part of the DRAGON project [ZLdS98], they have devoted the efforts to enhance database replication mechanisms by taking advantage of some of the properties of group communication primitives.

The previous researches have shown how group communication can be embedded into a database [AAAS97, PGS97, PGS98] and used as part of the transaction manager to guarantee serializable execution of transactions over replicated data [KA98, KA99].
They have also shown how some of the overheads associated with group communication can be hidden behind the cost of executing transactions, thereby greatly enhancing performance and removing one of the serious limitations of group communication primitives [KPAS99]. This work has proven the importance of and the need for a common understanding of the replication protocols used by the two communities.

Database replication methods are classified by two important parameters. One is when update propagation takes place (eager vs. lazy) and the second is who can perform updates (primary vs. update-everywhere) [WPS'00]. In order to guarantee the global serializability [RC96] of transaction execution, the distributed locking mechanism has to be necessitated in eager replication schemes. The transaction, in other words, cannot be committed until all the replicated copies of data changed by that transaction have been updated. Lazy scheme update a local copy, make the transaction for propagation of the changes, and some time after commit, the propagation transaction will be sent to the corresponding servers. In lazy schemes, hence, no other global concurrency control method is needed, since all the transactions can be committed locally. However the copies on the different site might be inconsistent temporarily in lazy schemes. The reconciliation is necessary to keep up the consistent database state in this case. Figure 2.3 shows the comparisons of properties in eager and lazy schemes.

<table>
<thead>
<tr>
<th>Features</th>
<th>Global Commit</th>
<th>Global Serializability</th>
<th>Response Time</th>
<th>Reconciliation</th>
<th>Deadlock Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eager</td>
<td>2 Phase Commit 3 Phase Commit</td>
<td>Strict</td>
<td>Slow</td>
<td>No need</td>
<td>High</td>
</tr>
<tr>
<td>Lazy</td>
<td>Local Commit</td>
<td>Relaxed</td>
<td>Fast</td>
<td>Need</td>
<td>Low</td>
</tr>
</tbody>
</table>

Figure 2.3 Comparisons of Eager and Lazy Scheme

In regard to who can perform updates, the primary-copy has to be updated first and then
the other copies can be updated in the primary-copy approach. The data can be only read at the node that has the non-master copy while the data can be read and written at master node that has the primary-copy. The update-everywhere approach allows any copy to be updated on the other hand. The more complex coordination should be involved than the primary-copy approach while the high speed access to the data can be achieved. Figure 2.4 summarizes the properties of primary-copy and update-everywhere schemes. The proposed replicated architecture is classified as the primary-copy and lazy schemes, in order to acquire the fast and predictable response time and simplify the coordination control for replication. Based on the two parameters, the replication management strategies of commercial products can be categorized as shown in Table 2.2.

<table>
<thead>
<tr>
<th>Who can perform updates</th>
<th>Owner</th>
<th>Global Locking</th>
<th>Flexibility</th>
<th>Reconciliation</th>
<th>Deadlock Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary-Copy</td>
<td>Master Node Only</td>
<td>No Need</td>
<td>Low</td>
<td>Need Partially</td>
<td>Low</td>
</tr>
<tr>
<td>Update-Everywhere</td>
<td>Every Node</td>
<td>Need</td>
<td>High</td>
<td>Need (Based on Timestamp)</td>
<td>High</td>
</tr>
</tbody>
</table>

Figure 2.4 Comparisons of Primary-copy and Update-Everywhere Scheme

From the performance perspective, the lazy and update-everywhere schemes are better than the eager and primary-copy scheme respectively. However the lazy and update-everywhere scheme have to adopt the global locking mechanism to control the concurrent update transactions. The global locking causes a substantial performance bottleneck in case of main memory database systems. Main memory database systems thus need the new paradigm for replicated main memory database.
### Table 2.2 Classification of Replication Management Strategies

<table>
<thead>
<tr>
<th>Who</th>
<th>Eager Scheme</th>
<th>Lazy Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary-Copy</td>
<td>INGRES</td>
<td>Sybase Replication Server</td>
</tr>
<tr>
<td></td>
<td>Oracle Synchronous Replication</td>
<td>IBM Data Propagator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Oracle Placement Strategy</td>
</tr>
<tr>
<td>Update-Everywhere</td>
<td>ROWA(Read-one/Write-all)</td>
<td>ROWAA(Read-one/Write-all available)</td>
</tr>
<tr>
<td></td>
<td>Oracle Asynchronous Replication</td>
<td>Oracle Symmetric Replication</td>
</tr>
</tbody>
</table>

29
3 Transaction Processing Techniques for Performance

As transaction processing techniques for fast response time and performance scalability, this section explains the multiversion with simple locking concurrency control method and PML-tree index method with partial key, maximum key, and link pointer in a node to consider cache coherence under multi processor systems.

3.1 MVSL: Multiversion with Simple Locking Concurrency Control

Traditional concurrency control based on locking mechanism has some disadvantages such that there is a lot of information for locking objects. This problem is more serious when a number of concurrent transactions is larger or when the space for meta-data is restricted, especially in case of main memory database systems. The traditional multiversion locking methods cause performance overhead since the locking information has to be managed in a transaction manager. Furthermore all the transactions that access a record must acquire the latch on the page that includes the page rather than a record itself. Hence, a new concurrency control paradigm is required to achieve the more concurrency and faster response time in the multi-user environment. Especially, the application that consists of a lot read-only transactions need the more opportunity to read the data item without conflicts and to reduce the locking overhead at the record level. Thus MVSL is proposed to support the more concurrency and the more scalability than traditional multiversion methods.

3.1.1 MVSL Protocol

The proposed concurrency control method provides the multiversion method with multiple granularities. The MVSL protocol has two phases to control the concurrent
transactions. A transaction has to acquire the table lock in order to access the record or the entire table. The MVSL protocol provides the five lock types based on the granularity of accessing data item. The detail lock types and lock compatibility matrix will be illustrated in section 4.1.3. The second phase is to control the concurrent access to the record by using the multiversion concurrency control method. For example, if a transaction T1 want to access the data item (record) x of table P, T1 have to acquire the shared lock on the table P first and then find the proper version of record x and read that version of x finally. The lock types for table and its protocol will be described in the section 3.1.3 and this section proposes the MVSL protocol for second phase.

The restriction of strong consistency from the traditional serializability can be relaxed by allowing multiple transactions to read and write different versions of the same data item, as long as each transaction sees a consistent set of versions for all the data items that is accesses. The multiversion technique for concurrency control is often used with timestamp ordering method rather than the two-phase locking method because there are too many lock information about each versioned data item. However the multiversion timestamp method is not proper to real-time environment and main-memory database system since the transaction that executed almost can be aborted by other transactions. In real-time applications, furthermore, concurrent transactions do not necessarily need serializability to produce a correct result. In many situations the result of a transaction can be regarded as correct as long as the result correspond to the real-world situation presented in the database even if the transaction does not serialize with other transactions.

This section proposes the multiversion with simple locking (MVSL) method that does not need to manage a lot of lock information and illustrates the correctness of multiversion with simple locking method. The proposed MVSL guarantees the relaxed serializability rather than the conventional strong serializability.
The basic idea of MVSL is that there is no need to acquire the read lock when the read operation is executed. It only keeps the lock information for write operation in the data item (record) itself. The notations of \( R_i(x_i) \) and \( W_i(x_i) \) denote the read and write operation on the data item with version \( x_i \) respectively and \( TS(T_i) \) to denote the timestamp of transaction \( T_i \). The following example schedule (1) can be produced by the proposed MVSL, but it is not allowed by the traditional multiversion method.

\[
W_0(x_0) \ R_2(x_0) \ W_1(x_1), \quad TS(T0) < TS(T1) < TS(T2)
\]  

(1)

In earlier methods, \( W_1(x_1) \) invalidates \( R_2(x_0) \) since the timestamp of \( T_1 \) is smaller than the \( T_2 \). Thus transaction \( T_1 \) or \( T_2 \) should be aborted and restarted. In the proposed MVSL, the schedule 1 is a correct schedule since they are regarded as the serialization order \( T_0 \rightarrow T_2 \rightarrow T_1 \). The next example schedule (2) is also can be regarded as correct schedule even if it is not 1-copy serializable.

\[
R_0(x_0) \ W_1(x_1) \ R_1(y_1) \ W_2(y_2) \ W_2(z_2) \ R_0(z_2), \quad TS(T0) < TS(T1) < TS(T2)
\]  

(2)

In schedule S2, the last read operation \( R_0 \) must read the latest version of \( z, z_2 \) which is produced by the transaction \( T_2 \). At this time the schedule (2) becomes a non serializable schedule because the serialization graph of schedule (2) has a cycle. The proposed MVSL support the relaxed serializability rather than the conventional strong serializability. Hence the schedule (2) is correct in the view of weak consistency.

The MVSL processes \( R_i(x) \) by first translating it into \( R_i(x_k) \), where \( x_k \) is the version of \( x \) with the largest timestamp less than or equal to \( TS(T_i) \) and the transaction that produces the version \( x_k \) must commit. If that version of \( x_k \) is locked by another transaction, the transaction \( T_i \) must wait until the lock of \( x_k \) is released. Otherwise, the read operation is processed normally. The MVSL also processes \( W_i(x) \) just by producing the new version \( x_k \).
It does not need to wait other transaction since the new version of x cannot be locked when it is created.

The MVSL has only one type of lock, exclusive lock for write operation and it is implemented into the record as a lock bit. Thus the extra space for lock information is not required and it is easily implemented. However since the read lock is not manipulated in MVSL, the relaxed serializability is guaranteed. The MVSL weak consistency is defined in the Definition 1.

**Definition 1** MVSL weak consistency. If each read operation can see a transaction-consistent database, the schedule is MVSL weak consistent schedule. In other words, each read operation can serialize with all update transactions that it sees even if the serialization graph of that schedule has a cycle.

To prove that any schedule produced by MVSL is correct, the Theorem 1 could be defined.

**Theorem 1** Every schedule $S$ produced by MVSL is MVSL weak consistent.

**Proof**: Let the $O_i$ and $O_j$ are data conflict operations. There are three kinds of conflict operations such as read-write, write-read, and write-write.

[case 1] $O_i$ : read operation, $O_j$ : write operation. The write operation can be executed without considering about the $T_i$’s commit and the version of data since $T_j$ just makes the new version of data item and it does not affect the transaction $T_i$. In this case, the serialization order $T_i \rightarrow T_j$ can be made. In the traditional multiversion method, if the $T_j$ have to commit before $T_i$’s commit, the $T_i$ must be aborted and restarted. However the MVSL does not consider the commit order of transactions with read-write conflict.
[case 2] \( O_i \) : write operation, \( O_j \) : read operation. In this case, the read operation can be performed after the commit operation of \( T_j \). The serialization order \( T_i \rightarrow T_j \) can be made since the \( T_j \) have to read the version that is created and committed by \( T_i \).

[case 3] \( O_i \) : write operation, \( O_j \) : write operation. In this case, the transaction \( T_i \) creates a new version \( x_i \) and the \( T_j \) creates the version \( x_j \). Strictly speaking, the \( x_i \) and \( x_j \) are distinct data items without data conflict. The serialization order \( T_i \rightarrow T_j \) can be made. The transaction \( T_j \) can commit before the \( T_i \)’s commit operation.

By case 1, case 2 and case 3, the write operations except the read operations can be serialized according to the strong serializability. In other words, the schedule without the read operation cannot have any cycle of serialization graph. Thus the schedule that is produced by MVSL can have a cycle that is made by read operations. Any MVSL schedule ensures the consistent database state but the read operation can be executed with the inconsistency temporarily. Finally, the MVSL schedule is MVSL weak consistent schedule.

3.1.2 Version Management

In locking mechanism, a read transaction can only have a lock for a certain data item (record) until an update transaction release the lock for that item. Multiversion concurrency control mechanism provides the several versions of the same data item. Hence the read transaction cannot be affected by the execution of update transactions.
Assigning Version Number

A transaction has a logical timestamp value that is used to determine its committed order in the entire set of transactions. Hence a record header should have a version number as illustrated in Figure 3.1. When an update transaction commits, version number is assigned a logical timestamp obtained by incrementing a global logical timestamp counter, called a global system commit number (GSCN). The logical timestamp that is assigned to a record is called as system commit number (SCN). SCN is stored in a memory word and the least significant bit means a delete bit and the most significant bit means a lock bit. Figure 3.1 shows a structure of SCN. The lock bit will be illustrated in next section 4.1.3. GSCN is always an even number of positive integer domain and should be incremented by two grades when the update transaction commits. The reason to do this is that the least significant bit is denoted as the deletion of record. If a transaction wants to delete the record, the transaction sets only the SCN as $SCN + 1$. The operation on the SCN is performed atomically.

When a record has one more versions, a transaction finds the version that is read or update by the transaction. The transaction ($T_i$) can find the version by comparing between the SCN of $T_i$ ($T_iSCN$) and the SCN of a visited version ($V_iSCN$). In other word, $T_i$ can find the version that has been satisfied that $T_iSCN$ is less than $V_iSCN$. And if $T_iSCN$ is greater than $V_iSCN$ and $T_i$ is same to transaction of $V_i$, $T_i$ can get the version, $V_i$. An Example about this procedure is depicted in Figure 3.2 and the detail is in Algorithm 3.1.1.
[Algorithm 3.1.1 Check Version of Relative Transaction]

INPUT:

Trans: a transaction ID of current update transaction
T_{i,SCN}: SCN value of the trans
Target: a record ID to update by trans

OUTPUT:

True or False

ccheckVersion(Trans, T_{i,SCN}, Target)

01: if (Target is not deleted) // delete bit of SCN is 0
02: if (Target SCN < T_{i,SCN})
03: if (not exist next version of Target)
04: return True
05:     get next version
06:     if (next version is not locked)
07:        if (SCN of next version > T_i,SCN)
08:           return True
09:     else // if next version is locked
10:        if (TID of Target equal to TID of T_i)
11:           return True
12:     end if
13:     else // Target SCN > T_i,SCN
14:        if (Target is locked and TID of Target equal to TID of T_i and
             not exist next version)
15:           return True
16:     end if // 02
17:     end if
18: return False

End of checkVersion

End of Algorithm 3.1.1

-------------------------------------------------------------------------------------------------------------------------------

Version Management

In most disk-based schemes [BC92], storage space for a certain number of versions is
pre-allocated on each page for efficient access, which could result in under-utilization of
storage space(e.g., each item on a page has a single version). In the proposed versioning,
on the other hand, space for versions is dynamically allocated as they are created.
Furthermore, since a database could consist of more millions of "cold" items that have
only version, and space is an important constraint in main memory databases, the goal was
to impose essentially zero overhead on data items due to versioning.

When a transaction wants to update a data item, a slot from a page should be acquired first. Each versioned data item is constructed as a linked list as depicted in Figure 3.3.

Figure 3.3 Version List Configurations

The transaction should manage the version list that contains the new version, old version and the deleted version. When a transaction commits, the committed SCN is assigned to the SCN of new version list and old versions and deleted versions should be managed by garbage collector. On the other hand, new version will be managed if the transaction has to be aborted. The version management procedure is described in Algorithm 3.1.5 for transaction commit and Algorithm 3.1.6 for transaction abort in section 3.1.4.

3.1.3 Locking Management

The locking unit of MVSL is the *record* and *table*. Traditional locking mechanism must have the lock information on each locking objects. The proposed method however does not manage the lock information on the record because the record already has the lock

38
information and the versioned record is managed. The lock mode of record is only exclusive (X mode) or not. It is very simple but the concurrency of transaction is high since the management cost of record lock information is very chiefly and a transaction can find the proper version by using the versioned record even though there is no lock wait list.

The actual implementation of the record locking mechanism can be optimized for memory residence of the objects to be locked. In a conventional system, locks are implemented via a hash table that contains entries for the objects currently locked. The objects themselves on disk contain no lock information. If the objects are in memory, may be able to afford a small number of bits in them to represent their lock status. As Figure 3.1, all records have lock information (e.g. lock bit). Update transactions in MVSL determine whether the record is locked or not, simply, by checking the lock bit of the record. If the record is locked, the update transaction registers itself into the waiting queue of the transaction that locks the record, and then it goes to waiting state itself. This procedure is depicted in Algorithm 3.1.2.

[Algorithm 3.1.2 Checking Whether Record Locked or Not]

INPUT:

Trans: a transaction ID of current update transaction
Target: a record ID to update by Trans

OUTPUT:

Success or Failure

recordLockValidate(Trans, Target)

01: while
02:   if (exist next version of Target)
03:     get next version
if (lock bit of next version is 0 or exit other next version)
return failure //partial rollback and retry the update
else
//Target already locked by other update transaction
release latch from page of Target by current Trans
put Trans into waiting queue of transaction owns new version
latching the page of Target
end if
else
break
end if
end while
return success
End of recordLockValidate
End of Algorithm 3.1.2

MVSL deals with only exclusive locks on records. If the first bit is set, then the object is locked, else it is free. If it is locked and the second bit is set, then there are one or more waiting transactions. The identity of these waiting transactions is stored in a conventional hash lock table. If a transaction wishes to lock an object, it first checks its lock bit. If it is not set, it sets it and is done with the locking process. Some type of test and set instruction must be used to avoid two transactions from setting the bit. Later on, if a second transaction wants to wait on the object, it sets the second bit on and adds itself to the list of waiting transactions in the lock table. When the original transaction releases its lock bit, it checks if the second bit is set. If not, there are no waiting transactions and it is done. If it is set, it must go through the conventional procedure to wake up a waiting transaction. Clearly, many details are omitted. However, the key point is that by far the most likely
situation with low contention is for a transaction to lock a free object, update it, and to release its lock before any other transaction waits for it. In this case, both the lock and the release can be done with a minimal number of machine instructions, avoiding the hash table lookup entirely.

The MVSL provides the various lock modes on the table lock granule to enforce the concurrency in which table operations such as drop table and add column may be concurrently occurred with normal transactions which are read and update of records. The locking information on the table has to be managed in the proposed concurrency control method because the transaction has to access the table before accessing the record. For example, the read transaction and update transaction acquire the IS mode lock for execution and the delete operation of entire table should have acquire the IX mode lock. Figure 3.4 describes the control structure of table lock information in the proposed lock

Figure 3.4 Control Structure of Table Lock Management
The update transactions, however, keep the lock for that version until it commits or aborts. Since the proposed method uses the conventional 2PL, it ensures the isolation property of transaction. Table 3.1 shows a lock compatibility table and Table 3.2 illustrates the lock conversion table of table locking for the proposed concurrency control.

Table 3.1 Lock Compatible Table

<table>
<thead>
<tr>
<th>Owner Requester</th>
<th>NONE</th>
<th>S</th>
<th>X</th>
<th>IS</th>
<th>IX</th>
<th>SIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>NONE</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>S</td>
<td>O</td>
<td>O</td>
<td>X</td>
<td>O</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>O</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>IS</td>
<td>O</td>
<td>O</td>
<td>X</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>IX</td>
<td>O</td>
<td>X</td>
<td>X</td>
<td>O</td>
<td>O</td>
<td>X</td>
</tr>
<tr>
<td>SIX</td>
<td>O</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 3.2 Lock Conversion Table

<table>
<thead>
<tr>
<th>Owner Requester</th>
<th>NONE</th>
<th>S</th>
<th>X</th>
<th>IS</th>
<th>IX</th>
<th>SIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>NONE</td>
<td>NONE</td>
<td>S</td>
<td>X</td>
<td>IS</td>
<td>IX</td>
<td>SIX</td>
</tr>
<tr>
<td>S</td>
<td>S</td>
<td>S</td>
<td>X</td>
<td>S</td>
<td>SIX</td>
<td>SIX</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>IS</td>
<td>IS</td>
<td>S</td>
<td>X</td>
<td>IS</td>
<td>IX</td>
<td>SIX</td>
</tr>
<tr>
<td>IX</td>
<td>IX</td>
<td>SIX</td>
<td>X</td>
<td>IX</td>
<td>IX</td>
<td>SIX</td>
</tr>
<tr>
<td>SIX</td>
<td>SIX</td>
<td>SIX</td>
<td>X</td>
<td>SIX</td>
<td>SIX</td>
<td>SIX</td>
</tr>
</tbody>
</table>
3.1.4 Concurrency Control Algorithms

The proposed method does not acquire the lock for read operation and insert operation. However, in case of update and deletion operation, the latch is temporarily required during the next version is created. Even though the checkpoint is performed without the latch on the page, WAL protocol is ensured so that the concurrency control can perform without latch on the page during the insert, delete, and update operations. Especially, the read operation is not required to be set the latch because the proper version of record can be found atomically.

The algorithms for the proposed concurrency control are described in follow.

[Algorithm 3.1.3 Update Transaction]

**INPUT:**
- Trans: a transaction ID of current update transaction
- Target: a record ID to update by Trans
- Value: new value to assign to the Target

**OUTPUT:**
- None

**update(Trans, Target, Value)**
01: making and initializing a new version
02: get pid of the new version
03: latching the pid
    // locking the target and assigning the new version by value
04: if (recordLockValidate(Trans, Target))
    // if locked, wait to release the lock
//locking new version, updating target and managing version

05:  set next version ptr of target to be RID of new version
06:  unlatching the pid
07:  copy value of Target to data of new version
08:  registering the page of new version into dirty page list
09:  registering new version into version list of Trans
10:  registering Target(old version) into version list of Trans
11:  return success
12:  else  //can’t locking the target at all
13:    unlatching the pid
14:    return failure
15:  end if

End of update

End of Algorithm 3.1.3

[Algorithm 3.1.4 Read Transaction]

INPUT:

  Trans: a transaction ID of current running transaction
  readSCN: a version number of current read transaction

OUTPUT:

  Target: a record ID to find

read(Trans, readSCN, Target)

01:  while
02:    if (not exist a next version)
03:      return target
04:    read next version and assign to Target
05: if (checkVersion(Trans, readSCN, Target)) is true
06:     return target
07: end while
End of read
End of Algorithm 3.1.4

[Algorithm 3.1.5 Commit]

INPUT:
Trans: a transaction ID to abort

OUTPUT:
None

commit(Trans)
01: increment GSCN //CommitSCN = CommitSCN + 2
02: set CommitSCN from GSCN value
03: set SCN of version list by CommitSCN
04: copy list of old and delete versions on Trans to garbage collector
05: set SCN of the version list in garbage collector by GSCN value
06: write commit log
07: release all locks in Trans
End of commit
End of Algorithm 3.1.5

[Algorithm 3.1.6 Abort]

INPUT:
Trans: a transaction ID to abort

**OUTPUT:**

None

**abort(Trans)**

01: rollback the actions of the Trans by undo(Trans)
02: write abort log
03: copy list of new versions on Trans to garbage collector
04: set GSCN value to SCN of the version list in garbage collector
05: release all locks in Trans

**End of abort**

**End of Algorithm 3.1.6**
3.2 PML-tree: Cache Conscious Tree Index

Main purpose of index techniques in disk-based system is to minimize the number of disk access. In a main memory DBMS, however, the computation time with minimal memory space is the key consideration point to design the index mechanism, because the whole data set resides in memory. There is no need to contain the actual data value in an index node. An index node just has a pointer to the data instead of real data value.

However, the conventional index mechanisms generally require the latch on the index node during the index tree traverse. It causes a significant bottleneck to access the index tree concurrently. The lock coupling problem while the index tree is modified moreover reduces the overall performance for the read operation on that index tree. The index version technique has been suggested to achieve the higher performance. It however causes the memory and timing overhead to manage the version of index node.

The T-tree index with index node versioning has the more memory efficiency and the less time of computation than B'-tree. The T-tree however can cause the performance degradation due to a lot of cache miss under multiprocessors since a node of T-tree has data pointers instead of actual data values and update transactions do many latching index nodes [LC92, LNT00]. Also index node versioning of the T-tree moreover sometimes requires more memory space and degrades the concurrency the tree operations. Hence the modified B'-tree, called PML-tree, is suggested that has the minimal partial key instead of the full key, a maximum key value, and link pointer in a node. The PML-tree provides the faster response time and the higher performance scalability with considering the cache conscious concurrency control on multiprocessor systems.

3.2.1 Index Structure
The access speed of cache is ten or more times faster than the main memory. The recent index mechanisms of MMDB are hence assumed to be fully loaded into cache memory. Because a node of T-tree just has the pointer to data value, when tree operations compare key values of index nodes, a lot of cache miss should occur. Thus the PML-tree is provided that has the partial key. As shown in Figure 3.5, the PML-tree mechanism provides the key comparison with partial keys and the child node pointers in an index node.

The PML-tree is a balanced multi-way search tree. Every node in the tree of order \( d \) contains \( m \) sorted boundary partial key values, where \( d \leq m \leq 2d \). A PML-tree puts all the child nodes of any given node into the child pointers of the node. There are two kinds of index node, internal and leaf node. An internal node has version number, set of \(<\text{partial key, row pointer, child pointer}>\), maximum key and a link pointer to the right sibling node. The version number is used when the node is updated and split. The maximum key means that is greater than the highest partial key included in a given node and less than or equal to the lowest partial key of just next sibling node if exists. All sibling nodes in PML-tree are linked to left direction by link pointer in a node.

A leaf node consists of a partial key value, pointer to the actual data record and a pointer to the right sibling leaf node. One of most important feature of PML-tree is that the two kinds of nodes have the partial key value. It is useful to use the memory space efficiently when the length of key value is long. It also reduces the CPU time and cache miss ratio since it does not need to read the actual record value when the index node does not have the key value. Furthermore the pointer to the right sibling node like Blink-tree is also maintained, so that other transaction can traverse the index tree even though the index structure is changed without lock coupling at the same time.
3.2.2 Operations on a PML-tree

The traditional B⁺-tree should acquire the latch or lock first before access an index node. It causes a lot of coherence cache miss. The index node versioning (INV) of T-tree also requires more memory space during the index structure modification. Index operations on PML-tree are similar to a B⁺-tree and Blink-tree. Hence this section explains the tree operations which are tree traversal, split and deletion of node, since PML-tree is revised for the cache coherence and high concurrency of the transactions.

Latch-free Traversal

This tree traversal technique enables a read-only transaction to concurrently execute with update transactions. This can be done by an extra field of index node for version number as shown in Figure 3.5. An update transaction first acquires the latch for the node, modifies the value of index node and then increases the version number. A read-only transaction checks the latch for that node. If the latch is not set, the version number is copied into local space and the transaction is executed. After the execution, the version number of that index node and the copied number should be compared in order to preserve the data consistency.

Algorithm 3.2.1 shows the pseudo code for the tree traversal. The find_next(node, key) primitive finds the next node to traverse. If the key is greater than the high key of the
node, this operation returns the link pointer. Otherwise, it returns the pointer to the appropriate child node to traverse down. The procedure read node is embedded into find_next on the procedure traverse of Algorithm 3.2.1. The while loop of the read node is removed by assigning the value of next to the variable node only if the next is computed from a node in a consistent state.

[Algorithm 3.2.1 Latch-free Tree Traversal]

**INPUT:**
- root: root node pointer of the tree index
- key: key value to search the target records
- node: the node to read or update during traverse the tree

**readNode(node)**
01: do
02: copy version number of the node to t
03: read the contents of the node
04: if (latched the node already)
05: continue the loop
06: while (not equal t and node.version)

End of readNode

**updateNode(node)**
01: acquire a latch on the node
02: update the content
03: increment version number
04: release the latch
traverse(root, key)
01: assign root to node
02: while (node is not a leaf)
03: copy node.version to t
04: if (latched the node)
05: go to 03
06: visit a next node using find_next(node, key)
07: if (equal to t.version and node.version)
08: assign next node to node
09: end while
10: return node
End of traverse
End of Algorithm 3.2.1

Left-most Node Split
Since latch free index traversal does not use lock coupling while traversing down a tree index, concurrent updaters may split the target child node of the traversal before the traversal reaches the child node. Moreover, since no latch is held while reading a node, concurrent update transactions may also split the node concurrently being read. To deal with this problem, the technique of using maximum key and a link pointer is suggested.

All split are done from the left to the right and maximum key and link pointer are added to each node. The maximum key is upper bound of the key values in the node and the link pointer is to provide an additional method for reaching a node and whether to follow the
link pointer or not can be determined by the maximum key. With this link pointer, since splits are done from the left to the right and each node has its maximum key and its link pointer to the right neighbor, all nodes split from a node are reachable from the node and the correct child node can be reached in the presence of concurrent splits of nodes.

Figure 3.7 shows an example node split flow of proposed tree and Figure 3.6 shows an example node split of typical tree without maximum key and link pointer. In the Figure 3.6, the transaction 2\((T2)\) can not find the key 4, since the node B including key 4 has been split into node D by transaction 1\((T1)\). This problem is caused by which a read node operation on the tree is traversing down without any latch on the node. Thus, \(T2\) re-traverses from the root node. To solve this problem, PML-tree provides a maximum key and a link pointer in a node. As shown Figure 3.7, by splitting the node B from T1, the key 4 was moved into node D just before T2 visits the node on step 3. So, T2 can not find the key 4 in node B, but T2 can visit the node D following the link pointer in node B and can find key 4 in node D.
1. Transaction 1: Insert 5
2. Transaction 2: Find 4

- Transaction 1: Find leaf node in the tree, and choose B

- Transaction 2: Find child node in A, and choose B

- Transaction 1: Insert 5 into B node, and split

- Transaction 2: Can't find 4 in B node because of splitting node B

Figure 3.6 Typical Node Split in the Traditional B’-tree
Lazy Node Deletion

The empty node without key is not immediately removed in the index tree. Such an empty node can be still used by read transactions without latch which is not yet committed. These garbage nodes are properly destroyed by the dedicated garbage collector. The garbage collector actually de-allocates the registered node when there are no index operations that can read the node. To determine whether there is any operation that can read the node or not, the algorithm originally the physical versioning is used.

3.2.3 Time and Space Analysis

In this section, the time performance and the space requirement of PML-tree are analyzed.
To simplify the presentation, there are assume that a version, a partial key, each pointer, and maximum key are same amount of 8 bytes, so the size of each element in a node, space $K$, is amount of 24 bytes. The $n$ denotes the number of leaf nodes being indexed and $c$ denotes the size of a cache line in bytes. The fan-out rate per node is denoted by $m$, which can be derived using $m = \frac{s}{K}$. The cache line size is assumed 256 bytes. Those parameters and their typical values are summarized in Table 3.3.

Table 3.3 Values and Parameters for Time and Space Analysis

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Notation</th>
<th>Typical Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of each element in a node</td>
<td>$K$</td>
<td>24 bytes</td>
</tr>
<tr>
<td>Number of leaf nodes</td>
<td>$n$</td>
<td>$10^5$</td>
</tr>
<tr>
<td>Size of a cache line</td>
<td>$c$</td>
<td>256 bytes</td>
</tr>
<tr>
<td>Size of a node</td>
<td>$s$</td>
<td>4096 bytes (4 KB)</td>
</tr>
<tr>
<td>Fan-out rate</td>
<td>$m$</td>
<td>168</td>
</tr>
</tbody>
</table>

The search time of PML-tree is affected by three factors, branching, key comparisons and cache misses [RR00]. The branching factor is $m = \frac{s}{K}$. It is same to branching factor of conventional B’-tree. The larger branching factor, the smaller the number of cache misses. The total key comparisons of PML-tree are the number of $\log_2{n}$. The number of cache misses is calculated by the factors that are the total key comparisons, key comparisons in a node, and multiple proportion of node size compared to cache line size. It leads to cache misses as follow:
The split cost is derived from the expected number of cache lines that need to be accessed during a split. PML-tree has two splits that is a smaller number of since the source and destination overlap for copies. This, in other words, means that PML-tree divides a node into two nodes. Since the node size is larger than cache line size in PML-tree, the accessed cache lines in a split are \( \frac{2s}{c} \), exactly 32.

The space requirement of internal node space is assumed all nodes are 70% full [RR00]. The internal node space cost of PML-tree is:

\[
\frac{4ns}{0.7(m - 2)(0.7m - 2)}
\]

So, when the number of leaf nodes is 10,000,000, the total size of internal nodes is 121.9 MB. In case of leaf node, the space cost is:

\[
\frac{2ns}{0.7(m - 2)}
\]

Thus the total size of leaf nodes is 704.9 MB.
4 Transaction Processing Techniques for Stability and Availability

As transaction processing techniques for stability and availability of database, this section explains the logging and recovery method and replication method. The logging is multilevel logging method to enhance the transaction performance and recovery provides latch-free fuzzy checkpointing for high concurrency of active transactions and one pass recovery method without analysis phase. The replication provides the lazy-flex replication scheme for replication performance and various replication configuration models.

4.1 Multilevel Logging and Recovery

The logging and recovery management is designed for recovery of any kinds of failure such as a media or system failure. The previous logging mechanisms do not support the adaptable logging way corresponding to the specific application environments. The checkpoint phase and analysis phase requires the most expensive cost in the recovery procedure. Thus it is the critical bottleneck point in the real-time applications. The efficient method can be suggested by using the relationship between transaction response time and durability. The proposed multilevel logging and recovery hence show the better concurrency during the recovery phase is performed.

To enhance the logging performance, the logging method provides the multilevel loggings related to transaction durability and uses the memory-mapped file that is directly mapped with one-to-one relationship to the memory buffer. The checkpoint is a latch-free fuzzy checkpoint method to avoid interference of live transactions. To fast restart recovery, the recovery method do not process the analysis phase to find recovery point in log file.
4.1.1 Multilevel Logging

The basic discipline of recovery management is the *WAL*(Write-Ahead Log) method in *ARIES* [MHL’92, Moh99] system. The proposed backup process can be performed during not only the off-line state but also the on-line database service state. For the recovery of database, the log manager generates the optimal log record and exploits the fuzzy and ping-pong check points. In this mechanism, two backup databases are manipulated and the current on-going transaction is not effected by the backup procedure. The *log flushing* has manipulating all kinds of log records and flushing the log record into the current log file on disk without any interference with execution of live transactions.

Log records are written into multiple log files for efficiency of recovery. As the proposed logging mechanism, two kinds of log buffer can be used by the recovery manager. The memory-mapped file is basically used as a log buffer. In this situation, the memory mapped file that is placed in the disk device with very slow I/O incurs overall poor performance. The effect of operating system overload, moreover, is directly reflected into the transaction performance. While the basic operation of transaction can be efficiently performed since the entire data is located in memory, the logging operation is very slowly completed because log records for that transaction are written into memory-mapped file which is used as log buffer. The overall performance, however, is significantly degraded due to the memory mapped log buffer. The log manager provides the memory buffer as a log buffer for alternative solution. Two kinds of log buffer, memory buffer and memory-mapped file, are supplied and user can determine which log buffer will be used based on the transaction durability.

Therefore, the multilevel logging is provided. There are four levels of logging for transaction performance and reliability of database state. Based on the transaction durability, the memory buffer or memory-mapped file can be used as a log buffer and the
synchronization techniques of log records are differently exploited.

*Logging level 1.* All log records are written into only the memory buffer, and any dirty page is not synchronized with the disk. If the database server should restart, any updates by the transaction execution cannot reflect into database state in this logging level. However the logging level 1 assures the consistent of in-memory database state by allowing the transaction rollback.

*Logging level 2.* The memory buffer is also used in logging level 2, but the logs must be synchronized with the log file by log flushing thread. The durability of committed transaction cannot be guaranteed in this logging level, because the transaction is declared as *commit state* before its commit log record is synchronized with the log file.

*Logging level 3.* The memory mapped file as a log buffer is used in the logging level 3. Because the written log records into this log buffer are flushed to the disk by operating system immediately, the log manager need not to flush the log buffer to disk log file in which log records are written into log buffer. Instead of, the all logs in log buffer are flushed by checkpoint manager periodically. Thus, the transaction durability in this logging level keeps from system failure.

*Logging level 4.* The logging level 4 also uses the memory mapped file. This level assures that the log records are written to disk log file before commit state of the transaction. In other words, since the consistent log file at disk level is the necessary condition for commit operation, the durability of committed transaction is ensured in level 4. There is trade-off relationship between the transaction durability and performance according to each logging level. The log manager hence provides the various durability levels of transaction as the system parameter which can be controlled by the database administrator.
The transaction manager also supports the partial rollback. Two kinds of savepoint are provided to support the partial rollback. Explicit savepoint is set up as the user request. Implicit savepoint, on the other hand, is set up by transaction manager for the system purpose. The major differences between implicit and explicit savepoint are described in Table 4.1.

<table>
<thead>
<tr>
<th>Differences</th>
<th>Explicit Savepoint</th>
<th>Implicit Savepoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Savepoint Name</td>
<td>Defined by User</td>
<td>NULL</td>
</tr>
<tr>
<td>Set/Rollback</td>
<td>by User</td>
<td>by System</td>
</tr>
<tr>
<td>Num. of points</td>
<td>Unlimited</td>
<td>One per Transaction</td>
</tr>
<tr>
<td>Log</td>
<td>Log Created</td>
<td>Without Log</td>
</tr>
</tbody>
</table>

4.1.2 Latch-free Fuzzy Checkpoint

The checkpoint is the part of recovery procedure that performs the writing the intermediate transaction results into a stable storage in order to minimize the cost of recovery. The proposed storage management model uses two database files for one database. They are used alternatively when the checkpoint is performed. This method is called it as a ping-pong checkpoint. The fuzzy checkpoint technique is proposed so that the transactions which are executed currently are not interfered when the checkpoint is performed. To minimize the cost of recovery, the checkpoint is maintained during the transaction execution time. The checkpoint that starting and ending checkpoint log is completely written into the log file is defined as complete checkpoint. Otherwise it is
named as *incomplete checkpoint*. Upon the *incomplete checkpoint*, recovery procedure is performed by using dirty page list and transaction list that were executed at the failure point. The ping-pong checkpoint is provided for the database file. Hence, the database status is guaranteed to be in correct state at the disk level until the previous checkpoint. Checkpoints are performed when the database starts, and *checkpoint thread* also perform the checkpoints periodically. Furthermore, the functionality *auto remove archive* enables us to wipe out the unnecessary log files.

One of the major advantages in the proposed recovery phase is to provide the higher concurrency since the information of active transaction does not be constructed when the checkpoint is performed. ARIES [Moh99] system makes the list of active transaction at the checkpoint. This causes the large amount of conflict between transactions and thus degrades the concurrency. The recovery manager can perform the undo phase by using only the least LSN of log files.

Another feature of recovery is that the dirty pages are reflected into database file on stable disks only at the checkpoint time. In MMDBMS, all data pages are in main memory and some of them are changed. If those dirty pages are not reflected into stable storage during the whole operation time, it is too risky and there are a lot of works for recovery. Hence all of the dirty pages are reflected into stable disk at the checkpoint time.

The dirty page list of log manager that is constructed at the checkpoint time is not removed from the system since there are two database files for ping-pong checkpoint per a database. The dirty page list can be removed when they are fully reflected into both database files. Algorithm 4.1.1 describes the algorithm of the fuzzy checkpoint without any latch.
[Algorithm 4.1.1 Latch-free Fuzzy Checkpoint]

INPUT:
before_dirty_page_list: dirty page list of complete checkpoint before
current_dirty_page_list: current dirty page list

OUTPUT:
None

checkpoint(before_dirty_page_list, current_dirty_page_list)
// restart_lsn is lsn(log sequence number) of restart recovery
// begin_checkpoint_lsn is lsn of begin log of current checkpoint
// begin_trans_lsn is lsn of transaction begin log

01: choice the database copy performed checkpoint
    // decision of restart_lsn
02:  if (exit active transactions)
03:      if (begin_checkpoint_lsn less than equal the least begin_trans_lsn)
04:          set restart_lsn by begin_checkpoint_lsn
05:      else
06:          set restart_lsn by the least begin_trans_lsn
07:      end if
08:  else
09:    set restart_lsn by begin_checkpoint_lsn
10:  end if
11: write checkpoint begin log including restart_lsn
12: logging before_dirty_page_list to log file
    //writing dirty the pages without any latch
13: write dirty pages in before_dirty_page_list to the database copy
14: write dirty pages in current_dirty_page_list to the database copy
15: flushing log buffer
16: copy current_dirty_page_list to before_dirty_page_list
17: write end checkpoint log
18: write restart_lsn to log anchor file //for restart recovery
End of checkpoint

End of Algorithm 4.1.1

-------------------------------------------------------------------------------------------------------------------------------

4.1.3 Restart Recovery

The proposed logging and recovery method provides a log anchor and number of log files in disk. The log anchor includes the log file number and restart lsn to need database recovery. The log file number in log anchor denotes the first log file to need recovery start and include the log pointed to the restart lsn.

In ARIES system [MHL'92], recovery procedure consists of analysis phase, redo phase and undo phase. Dirty page list and uncommitted transaction list are generated at the analysis phase, the data value of all the committed transactions is restored at the redo phase, and finally the partial work by uncommitted transaction is invalidated at the undo phase. In the proposed recovery method, the recovery procedure is divided into only redo and undo phase. There is no analysis phase, since the dirty page list has been already written to log file at the checkpoint time and the recovery start point in the log file has been stored in log anchor at the latest checkpoint time. The information of uncommitted transactions is collected on redo phase. So, the restart recovery could be executed fast as well as complete. The restart recovery algorithm is explained at the Algorithm 4.1.2.

-------------------------------------------------------------------------------------------------------------------------------

[Algorithm 4.1.2 Restart Recovery]
restart()
01:    choose a database file to recover
02:    load all data into memory from the database file
03:    redoAll()
04:    undoAll()
05:    checkpoint()
End of restart

redoAll()
01:    get restart LSN from log anchor file
02:    while (LSN is invalid)
03:        read a log record pointed by LSN
04:            switch (type of the log record)
04:                case 'dirty page list'
05:                    register the list to dirty page list of log manager
06:                case 'begin of transaction'
07:                    register the transaction to transaction table and
08:                        listing it to log manager
08:                case 'commit' or 'abort'
09:                    do end action of the transaction
10:                case 'update'
11:                    execute redo action for the log record
12:                        register the updated page to dirty page list
13:                case 'CLR'
14:                    execute undo action for the original record of CLR
15:                case 'end of log file'
16:                    increment lfsn of LSN
17:                        set offset of LSN to 0
18:            end switch
20: set LSN to read next redo log record
21: end while
22: set the information of the log file written the last log record
End of redoAll

undoAll()
01: while (exist undo transactions in transaction table)
02: get a Tx from transaction manager
03: get a first undo LSN of Tx
04: while (exist log record for undo LSN)
05: read a log record indicated by undo LSN
06: execute undo action for the log record
07: set next undo LSN from previous lsn in the log record
08: end while
09: delete Tx from transaction table
10: end while
End of undoAll

End of Algorithm 4.1.2

---------------------------------------------------------------------------------------------------------------
4.2 Lazy-flex Replication

Replication is used mainly for fault tolerance purpose in general distributed systems. In cooperative database system environment, on the other hand, the replication is utilized as a way to increase the performance as well as the availability. Current conventional DBMS applications require the high availability as the necessary function of DBMS. Although a large number of replication mechanisms and protocols have been proposed, few of these methods have ever been used in commercial products [KA00, WPS’00, GHOS96]. In the main memory DBMS, furthermore, these replication methods are not adequate because of predictable response time.

Since log-based replication is faster than the query-based[AAAS97], the lazy-flex replication uses the log-based replication rather than the query-based scheme since the application still requires the fast response time during the replication phase. The lazy-flex replication method is based on the traditional lazy scheme. In this scheme, there may be data inconsistent state temporarily, but it can be controlled by a data synchronization techniques. The lazy-flex replication provides the various data synchronization protocols corresponding to the applied replication models. Hence the extra procedure for data consistency is not needed in the proposed lazy-flex replication method.

4.2.1 Replication Model

As illustrated in section 2.4, the proposed replication scheme is classified as the lazy and the Flex which is a combination of primary-copy and update-everywhere, in order to acquire the fast and predictable response time and flexible configurations of replication model and simplify the coordination control for replication. In the traditional update-everywhere scheme, there may be data inconsistency temporarily when some kinds
of replication failure occur. To resolve this problem the proposed lazy-flex replication scheme provides the new paradigm for replication mechanism for data consistency. Furthermore, the conventional lazy scheme provides the better performance but it cannot have the immediate data synchronization between replicated nodes. The storage management system thus provides a new replication mechanism that is based on lazy and flex scheme with various efficient data synchronization protocols and a data conflict resolution mechanism.

The lazy-flex replication scheme basically replicates the updates by point-to-point and transfers the updates to N-way for all replicated nodes. Thus, the replication topology is consisted to network structure if all nodes are applied to update-everywhere. Figure 4.1 shows the full replication topology of the lazy-flex replication scheme.

The lazy-flex replication scheme can flexibly set up the various replication models according to the practical use environment of database, that are primary-copy model, update-everywhere model and combination of those models. It, exactly, provides the four replication models for supporting the various time-critical and reliable database applications. A replication model is categorized by the role of nodes that are participated in replicated database system. Three types of replication node are proposed as described in Table 4.2.
Table 4.2 Node Types in Lazy-flex Replication

<table>
<thead>
<tr>
<th>Node type</th>
<th>Role description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master node</td>
<td>The master node commits own transactions and propagates its updates to every replicated node and does receive any update from other replicated nodes.</td>
</tr>
<tr>
<td>Slave node</td>
<td>The slave node receives the propagated updates from the master nodes and then reflects the updates into own local database. In normal operation state, this kind of node cannot service the database request.</td>
</tr>
<tr>
<td>Propagator node</td>
<td>The propagator node receives the updates from the master nodes and propagates the received updates to other slave nodes simultaneously.</td>
</tr>
</tbody>
</table>

Figure 4.2 Replications Models in Lazy-flex Scheme

Based on the node types in the Table 4.2, four types of configurations of replicated nodes are illustrated in Figure 4.2. The basic replication model is *master-slave* model depicted
in Figure 4.2(a), which is the typical model for fault-tolerance in the conventional
distributed systems. All the database requests are handled at the primary node(N_i) and
their updates have to be propagated into the slave node(N_j). When N_i → N_j, i ≠ j, is
denotes the primary-slave replication, there is not allow N_j → N_i at all. The slave node
does not manipulate any kinds of database updates except read operation in normal
operation mode and is only ready to immediately take over in case the master server fails.
After the failure has completely recovered, the slave node can be played the role of
primary node. In other words they can change over their roles each other.

The more flexible model for application development, master-master model is shown in
the Figure 4.2(b). This replication model enables to develop the load-balanced
applications as well as the highly available applications. When N_i → N_j, i ≠ j, denotes the
primary-slave replication, there is allowed N_j → N_i all the time. To design the
load-balanced applications, transactions should be categorized into several groups so that
the distinct group of transactions can be executed at the separate node. The updates of
each group of transactions should be propagated to other active nodes where other groups
of transactions are executed. A master node plays the both roles of primary and slave node
at a time. Each local database in the master node is in the identical consistent state after the
transaction has been committed. Although the proposed replication scheme is master
update, it also provides every-where for various replication models. In this case, there
exists data inconsistency between replicated databases since the replication model do not
support any global coordination mechanism. The replication management thus has the
data conflict resolution methods based on timestamp and value-based resolution.

The master-multi slave model illustrated in Figure 4.2(c) is for the applications that are
highly sensitive to the performance and availability. This model consists of one primary
node and two or more slave nodes. When N_i → N_{j,k}, i ∉ {j, j+1, j+2, ..., k}, denotes the
primary-multi slave replication, there is not allowed N_{j,k} → N_i at all. So there is no nay
circle among the replicated nodes. The basic replication protocol of master-multi slave model corresponds to the master-slave node, but it is differentiated from the number of replication and its performance. It therefore can acquire the higher availability than other models but pay the expensive cost for multiple replications.

For the efficiency of multiple replications and supporting the reasonable response time of main memory DBMS, the *propagator* can be adopted, that is dedicated to broadcasting the primary node’s updates to the all slave nodes as Figure 4.2(d). When \( N_i \rightarrow N_j \rightarrow N_{k..n}, \ i \neq j \) and \( i \) or \( j \not\in \{k, k+1, k+2, \ldots, n\} \), denotes the primary-multi slave with propagator replication, there is not allowed \( N_{k..n} \rightarrow N_i \) and \( N_{k..n} \rightarrow N_j \) at all. The replication can also construct the hierarchical replication model based on the number of slave nodes and place the dedicated propagator into an internal node of the hierarchical replication model.

### 4.2.2 Replication Manager Architecture

The proposed replication manager is placed between query processor and storage manager as illustrated in Section 3. The replication manager was implemented with the thread architecture. The major components of replication manager are *manager*, *sender*, *receiver*, and *executor*. The sender and receiver thread has the responsibility for sending and receiving the replicated data called XLOG. The manager takes the charge of arbitration for replicated data between sender and receiver. A sender sends the replicated data to remote replication node through the manager in the receiver node as depicted in Figure 4.3. Hence there is one pair of the sender and receiver for a replicated node and they are dedicated to service for the replication only between those nodes.
Figure 4.3 shows the database replication system that consists of a master node and one replicated node which also is master. In this example system, in a node, two pairs of sender and receiver are existed between the master node and each replicated node. In the master node A, the sender is created and the receiver is created at the replicated node B. The synchronizing of replicated data is managed by the managers in node A and node B. There is one manager in a node and it has the control over its own receiver. Under the control of manager, the replicated data is transferred and completely synchronized.

4.2.3 Various Synchronization Protocols

The basic idea of the proposed replication management is based on the conventional transaction logs. The proposed replication method based on the log records does not need that the active server is not required to keep the information on the database updates. Replication manager transfers the transaction log from the last committed transaction’s log after the failure recovery because the transaction log is always kept in each server.
Only the change log, $XLOG$, is additionally maintained in order to propagate the updates to a remote server and it is plainly constructed from the general DBMS transaction log. $XLSN$ is the sequence number for $XLOG$ and it refers to the point of transaction log that is not yet reflected into the remote replicated server. The replication manager gathers the transaction log records from the $XLSN$ to the current log record pointer and then composes the $XLOG$ transaction from those log records. The supplementary information for database replication is not obliged to propagate the updates and keep up the consistent replicated state.

The proposed replication mechanism is implemented by the replication manager thread ($Manager$), the sender thread ($Sender$) and the receiver thread ($Receiver$). Those threads are operated cooperatively in order to reflect the update of another replicated server. Figure 4.4 illustrates the replication protocol and communication flow.

![Figure 4.4 Coordination Flow between Sender and Receiver](image-url)

When the replication is started, the sender thread is created and then it requests a
handshake connection for propagating the update to the replication manager thread at the remote node (Figure 4.4 (S1)). At the next stage, the replication manager thread creates the corresponding receiver thread as shown Figure 4.4 (S2). When the receiver thread receives the handshake connection request safely, it sends the ACK message to the replication manager and sender thread in sequence.

When the transaction have been committed at the master server, the sender thread makes the update transaction based on its own the change log, XLOG, of the transaction. The sender node should manages the logs and the starting and current point of transaction log, the point(XLSN) of change log so as that all the updates are ensured to reflected to remote node. The transactionized updates are sent to the replication manager and receiver thread in sequence as depicted in Figure 4.4 (S5) and (S6). When the sender thread, however, cannot make a handshake connection to the replication manager thread because of network failure (Figure 4.4 (F-A1)), the connection retry occurs at the given system parameter as illustrated Figure 4.4 (F-A3). When the receiver thread cannot reply to the replication manager or sender thread, the overall replication procedure should be aborted immediately as shown in Figure 4.4 (F-B1) and (F-B2). On the other hands, when the sender thread receives the NACK message for the reason of logical error such as the schema conflict (Figure 4.4 (F-A2)) even though the connection was established completely, the sender thread should be terminated(Figure 4.4 (F-A4)).

The 2 phases or 3 phases commit method in traditional distributed system use the synchronous techniques so that they cannot be used in MMDBMS. Asynchronous protocol for replication must be adaptable to MMDBMS because of performance improvement. Asynchronous protocol however may be made a database inconsistent state when a failure occurs. In order to avoid this problem, the lazy-flex replication method provides the three kinds of synchronization transfer protocols as illustrated in Figure 4.5.
Asynchronization protocol. The replication log can be transferred without confirmation of reflection updates into remote databases. When a transaction’s state becomes commit state, its update is reflected into local database and then the related log records can be sent to the replicated database. This protocol enhances the overall performance but the database state of replicated nodes can be inconsistent state with the local database. After the recovery phase, the transaction’s log that was not sent to replicate node should be transferred to that node so that the replicated database becomes a consistent state. The local node should recognize whether the replicated database reflects the related log records into its database state or not. The local node also keeps the points to the log records that are already reflected into the replicated node.

Semi-Synchronization Protocol. In this protocol, a transaction can only commit after that the replication log records are perfectly ensured to send to the replicated node. In order to avoid the disadvantage of asynchronization protocol, the update log is first sent to the replicated server before the commit of transaction. When the failure occurs in the local node, the database state of replicated node is ensured to reflect all updates of local database. The replicated node can be ready to provide the database service promptly before the local node executes to recover the failure. But if the failure occurs before the replicated node does not receive the update log, the replicated database can be consistent state after the recovery of failure.

Complete Synchronization Protocol. A transaction can commit only if the update log is fully reflected into the replicated database. In this protocol, the replicated database is ensured to keep the consistent state, but it causes substantial performance bottleneck.
4.2.4 Algorithm of Sender and Executor

The replication procedure for the proposed protocol is described at the pseudo code level as Algorithm 4.2.1.

---

**Algorithm 4.2.1 Synchronization between Sender and Receiver**

```plaintext
sender()
01: try to connection to receiver node
02: if (connected successfully)
    03: set the current log point as the starting point of replication
       // Make the update transaction based on change log
04: while (do not request the termination of replication)
```

Figure 4.5 Synchronization Flow of Update Transaction

---
if (equal to current log point and replicated log point(XLSN))
continue
make the update transaction from the log record
switch (type of log record)
case 'transaction start log'
    register the update transaction to the replication manager table
    break
吲 case 'transaction end log'
    store the end log records into the transmitting buffer
    unregister the update transaction from the replication manager table
    break
吲 case 'update log'
    store the update records into the transmitting buffer
  end switch
increment the point of change log(XLSN)
// sending XLOG buffer to receiver of replicated node
if (buffer is full)
    send the transmitting buffer to the replication manager
end while
terminate the replication protocol
End of sender

executor()
// Receive and reflect the update transaction
while (connected to replication manager)
    read XLOG from the received update transaction
03:     switch (types of XLOG record)
04:         case 'transaction begin log'
05:             register the update transaction to the replicated
              transaction table
06:             break
07:         case 'transaction end log'
08:             unregister the update transaction from the replicated
              transaction table
09:             break
10:         case 'update log'
11:             reflect the XLOG record into the local database
12:     end switch
13:     end while

End of executor

End of Algorithm 4.2.1

---------------------------------------------------------------

4.2.5 Data Conflict Resolution

Database replication methods are classified by two important parameters. One is when
update propagation takes place(eager vs. lazy) and the second is who can perform
updates(primary vs. update-everywhere) [WPS’00]. In order to guarantee the global
serializability [RC96] of transaction execution, the distributed locking mechanism has to
be necessitated in eager replication schemes. The transaction, in other words, cannot be
committed until all the replicated copies of data changed by that transaction have been
updated. Lazy schemes update a local copy, make the transaction for propagation of the
changes, and some time after commit, the propagation transaction will be sent to the corresponding servers. In lazy schemes, hence, no other global concurrency control method is needed, since all the transactions can be committed locally. However the copies on the different site might be inconsistent temporarily in lazy schemes. The reconciliation is necessary to keep up the consistent database state in this case. In regard to who can perform updates, the primary-copy has to be updated first and then the other copies can be updated in the primary-copy approach. The data can be only read at the node that has the non-master copy while the data can be read and written at master node that has the primary-copy. The update-everywhere approach allows any copy to be updated on the other hand. The more complex coordination should be involved than the primary-copy approach while the high speed access to the data can be achieved.

The proposed replicated architecture is classified as the primary-copy and lazy scheme, in order to acquire the fast and predictable response time and simplify the coordination control for replication. Hence, the data conflict can occur between the replicated servers since the replicated data copies might be in temporarily inconsistent state. The two-phase and three-phase commit methods guarantee the global serializability in the conventional distributed system. However, these mechanisms cannot be adopted because of unpredictable response time and the feature of deferred replication scheme.

The lazy-flex replication method therefore provides the functionality of resolving those data conflicts by using the user’s conflict synchronization policy. The replication manager should designate the one of replicated servers as the master server and the others are considered as slave servers. In master-slave environment, the Table 4.3 summarizes the possible inconsistent states.

The replication manger monitors the above inconsistent states and resolves the conflict values based on the corresponding synchronization policies. According to the user’s
conflict synchronization policy described in Table 4.4 was established by the system administrator or DBA, the record in slave DB can be automatically adjusted as the correct record value in master database based on the user’s conflict synchronization policies.

Depending on the conflict synchronization policies, the replication manager always trace the replicated data copies and adjust the conflict value to the consistent state in the proposed replication system.

<table>
<thead>
<tr>
<th>Inconsistent States</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOSX</td>
<td>In regard to the value of primary key, there is a certain tuple in master DB but not in slave DB</td>
</tr>
<tr>
<td>MOSO</td>
<td>In regard to the value of primary key, there is a certain tuple in both master and slave DB, but they are not the same</td>
</tr>
<tr>
<td>MXSO</td>
<td>In regard to the value of primary key, there is a certain tuple in slave DB but not in master DB</td>
</tr>
</tbody>
</table>
Table 4.4 Data Conflict Resolution Policies

<table>
<thead>
<tr>
<th>Resolution Policy</th>
<th>Resolved Conflict</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SU Policy</td>
<td>MOSO</td>
<td>update the inconsistent value of slave record as the value of record in master DB</td>
</tr>
<tr>
<td>SI Policy</td>
<td>MOSX</td>
<td>insert some tuples into the slave DB, which are not in slave DB but in master DB</td>
</tr>
<tr>
<td>MI Policy</td>
<td>MXSO</td>
<td>insert some tuples into the master DB, which are not in master DB but in slave DB</td>
</tr>
<tr>
<td>SD Policy</td>
<td>MXSO</td>
<td>delete some tuples from the slave DB which are not in master DB but in slave DB</td>
</tr>
</tbody>
</table>
5 Performance Evaluations

5.1 Performance Evaluation Environment

5.1.1 Storage Management System for Evaluation

To evaluate the proposed methods that are MVSL, PML-tree, latch, multilevel logging and recovery, and lazy-flex replication, the storage manager adopted those methods is used. The storage manager is adapted to the commercial MMDBMS, ALTIBASE. As depicted in Figure 5.1, its major sub-components are concurrency control manager, recovery manager, index manager, transaction manager, memory manager, log manager, and replication manager [JLB03, JLB04]. And the storage manager has two database backup files and the multiple log files. The sub-components on shaded boxes are focuses to study of this thesis in section 3 and 4. The sub-components of the storage manager are described as follows abstractly.

Memory Management. The memory, the key storage space consists of persistent space and temporary space. The persistent space contains the physical database table and meta-data for system catalogs. This area is reflected into backup database. The temporary space, however, is not required to perform the backup procedure. The data that is not necessary to backup, such as index data, is located in temporary space. When the data is loaded into memory from disk, the index data is newly constructed. In proposed concurrency control method, the aged data that never be accessed cannot reside in memory. For the efficient memory management, garbage collector performs the revoking procedure for the old version data and then the acquired space is returned into the free memory space. The entire physical database is loaded into memory space after database started. The great part of memory data actually is not used for whole time of database service. The storage
manager hence provides the selective loading facility to maximize the convenience and utilization of memory space. Main memory space sometimes is not sufficient to operate the given user database operation. In this situation, the needless user table should be out from the memory into disk and then the obtained available space can be returned to free memory space. The functionality of database compaction is possible to perform the database compaction on the individual user table unit.

![Figure 5.1 Storage Manager of ALTIBASE](image)

**Transaction Management.** The further performance improvements and concurrency degrees can be obtained for transaction management by employing the proposed concurrency control method, MVSL (multiversion with simple locking). Particularly, applications can get more efficiency in which transactions are existed read-only and update extremely. The proposed transaction management method eliminates the unpredictable waiting for a data item that is a main disadvantage of conventional locking mechanisms. Transaction manager allows the data resource to be of various sizes and defines a hierarchy of data granularities. The small granularity of data (e.g.,
record) can be controlled by multi-version concurrency control while the large granularity of data (e.g., table) is managed by the lock-based method. The higher concurrency degree can be gained by applying the separate control schemes on the different level of granularity hierarchy. To skillfully manage the main memory, the transaction manager can identify and destroy the unnecessary aged version that is generated by update transactions. It can execute the number of transactions concurrently, and provides several properties related with the transaction execution with each isolation level e.g., consistent read, repeatable read, no phantom read.

Index Management. The purpose of index mechanism in DRDB is to minimize the number of access to disk. Main goal of index manager is to minimize the computational time of CPU with the least memory space. The great part of CPU time is wasted for waiting the memory access due to the cache-miss. To achieve the goal, the cache conscious concurrency control, in order to minimize the cache-miss ratio. The concurrency control for index node in traditional B+-tree provides the low scalability because a lot of cache-miss caused by the latch and lock. So, the storage management system provides an improved B+-tree, called PML-tree, which has a minimal partial key, a maximum value and link pointer in a node. The proposed index adapts the tree operations which are considered the cache consciousness to improve the overall cache-hit ratio under multiprocessor platforms.

Logging and Recovery Management. Log records are written into multiple log files for efficiency of recovery. To keep the reliable database state, transactions have the four durability levels corresponding to the significance of data. Depending on the durability level, the recovery manager utilizes the memory buffer or memory mapped file as its log buffer, and different log synchronization mechanisms. The recovery manager, furthermore, supports the three types of logging level based on the importance between transaction performance and consistency of data. The basic discipline of recovery
management is the \textit{WAL} (Write Ahead Log) method in \textit{ARIES} [MHL'92, Moh99] system. The recovery manager supports the on-line hot-backup and multiple log files. The proposed backup process can be performed not only at the off-line state but also at the on-line database service state. For the recovery of database, the storage manager generates the optimal log record and exploits the check points. In this mechanism, two backup databases are manipulated and the current on-going transaction is not effected by the backup procedure. The \textit{log flushing} has the responsibility of manipulating all kinds of log records and flushing the log record into the current log file on disk without any interference with execution of live transactions.

\textit{Replication Management}. The replication manager is placed between query processor and storage manager, since the basic replication mechanism is performed based on the update log and its transactional execution. The proposed replication manager supports the point-to-point replicated model and the replication is basically achieved by executing the log-based propagated update transaction. The remote replicated server analyzes the received update transaction and makes the execution plan for it. The replication manager also provides the network topology in order to support N-way replication.

\subsection*{5.1.2 Platforms and Workload for Evaluation}

An extensive set of actual experiments has been analyzed the behavior of the proposed method with respect to the settings of various design choices such as the concurrency control method, the index mechanism, , logging and recovery, and the replication model. The experiments present the comparative results under the variation of the adjustable system parameters. The main performance metric is the number of transactions per second (TPS), that is, the number of transactions that can be executed in a second. All experiments were
performed on HP 11.11 platform with 4 CPUs and 4G bytes of memory. The experimental environment is described in Table 5.1.

Table 5.1 Environment for Experiments

<table>
<thead>
<tr>
<th>Platform</th>
<th>HP RP5470</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System</td>
<td>HP.UX 11.0</td>
</tr>
<tr>
<td>Number of CPU</td>
<td>4(750Mhz)</td>
</tr>
<tr>
<td>Memory Size</td>
<td>4G Bytes</td>
</tr>
<tr>
<td>Number of Records</td>
<td>10,000 - millions</td>
</tr>
<tr>
<td>Number of Concurrent Users</td>
<td>1 - 64</td>
</tr>
<tr>
<td>Transaction Durability Level</td>
<td>3(persistent)</td>
</tr>
</tbody>
</table>

Most of experiments use four types of transactions such as *select, insert, update, and delete transaction*. The target table consists of total 20 attributes of various data types, such as *number, real, char and varchar*. *Update transaction* replaces the 17 attributes values per a tuple, and *insert transaction* performs the insertion of the complete tuples with 20 attributes into the table. All attributes values are fetched for the target tuple in *select transaction*. *Delete transaction* executes the deletion for one tuple with the search condition on the indexed key attribute.

The number of concurrent users in Table 5.1 is the number of applications (shooters) to evaluate the proposed methods. The performance evaluation model is that each application continuously shoots the transactions into the storage management system as Figure 5.2.
5.2 Performance of Concurrency Control Method

The proposed multiversion with simple locking (MVSL) concurrency control method enhances the concurrency of transactions, since the read transactions have any latching to the data and the update transactions use simple locking mechanism for record level locking. To evaluate the performance of MVSL, the workload has been executed by varying read and update ratio, e.g., the read-only transaction ratio has been changed from 0% to 100% by increasing grade 10. Figure 5.3 shows the result of the experiment. As shown the result, MVSL is better concurrency performance than the conventional 2PL [MHL’92] and MVCC [RSB’97]. The reason of the good performance is that MVSL reduces the locking management overhead for record level and can not any latching during read to a given version.

Figure 5.4 shows the overall performance of MVSL. This experiment can be analyze the performance in which are simultaneously executed the various combination of transactions, read-read, read-write and write-write. The purpose of the experiment is to evaluate how fast a version is created and how minimize the contention among the various transactions under MVSL.
Figure 5.3 Comparisons of Concurrency Control Methods

Figure 5.4 Concurrency of Various Transactions Combination on MVSL
To evaluate the performance of MVSL, the workload has been executed by on frequency of read 70% and write 30% which the frequency distribution of transactions is derived to real environment of database services on communication area such as HLR system. The evaluation was exploited by increasing the concurrent users in which are from 1 to 8, 16, 32, and 64. Figure 5.5 shows the result of the experiment. As shown the result, MVSL is better concurrency performance than the 2P and MVCC. While the performances of MVSL and MVCC are degraded at 6 users, the 2PL is 4. The reason of the result is that MVSL and MVCC reduce the contention of read transaction by using multiversion concurrency control, but the 2PL causes the lock contentions between read and update transactions. The reason of the better performance of MVSL than MVCC is that MVSL reduces the locking management overhead for record level.
5.3 Performance of Index Technique

The storage management system provides the PML-tree as described in section 3.2. PML-tree shows the comparative performance result with respect to the different index mechanisms. Figure 5.6 demonstrates the index performance of the PML-tree and T-tree INV [JLB 03] as the number of index increases. This experiment was performed under the single user environment and the data type of index was a char. In the cases of update transactions, the evaluated TPS decreases gradually as the number of index increases. Since the overhead of memory allocation for T-tree index increases, it requires more execution time. The Read transaction is not affected by the number of index directly because it does not require the modification of index structure. In both cases of read and update transaction, the proposed PML-tree shows the better performance than T-tree INV.

![Figure 5.6 TPS for Increasing the Number of Indexes](image)

Figure 5.6 TPS for Increasing the Number of Indexes
As illustrated in section 3, the T-tree index structure was adopted by several commercial MMDB because of better performance than the conventional B-tree. The T-tree however does not provide a better performance than the B'-tree because of the node versioning and locking for concurrent access to the index structure. Even though the modified T-tree with INV without latching and locking for read operation, the PML-tree index outperforms the T-tree INV index because the PML-tree index in MMDBMS has the partial key and cache conscious tree operations. Figure 5.7 plots the T-tree with INV and PML-tree index performance with respect to the several types of transactions. In case of single user environment, both index mechanisms show the almost same performance. However as the number of concurrent users are larger, the TPS of PML-tree is better than the T-tree with INV.

Figure 5.7 TPS of PML-tree and T-tree INV for Concurrency
5.4 Effect of Transaction Durability on Multilevel Logging

There is a trade-off relationship between the durability and performance. In order to ensure the perfect durability of database, the more execution time should be paid since the log records have to be reflected into persistent disk. According to the characteristic of application, the transaction durability can be adjusted to get the higher performance. The set of transaction with the various levels of transaction durability was experimented according to each logging levels. Figure 5.8 shows the TPS of read and update transaction with various transaction durability under 64 concurrent transactions environment. Obviously, read transaction shows the better performance than update transaction since it does not need to generate log records. It hence is not significantly affected by the durability level. Update transaction however have to write log records into memory mapped file and moreover those records should be synchronized with the persistent disk in the durability level 3 and 4. As shown Figure 5.8, TPS of update transaction steadily degrades as the durability level increases. Figure 5.9 also shows the TPS of transactions with respect to the number of concurrent transactions. The level 3 is sufficient to ensure the data persistency in general DBMS applications and the level 1 and 2 is required to the time-critical applications.
Figure 5.8 Performance of each Logging Level

Figure 5.9 Transaction Durability vs. Scalability
To study the effect of transaction processing on checkpointing time, the performance is evaluated in the transaction distribution in which the ratio of read transactions is 70% and the ratio of write transactions is 30%. The performance variations as the increasing concurrent user are experimented when the checkpoint phase is executed or not. In order to ignore the effect that comes from the dirty page flushing and logging of ongoing transactions, the database backup file and log file are separated. Figure 5.10 show the experiment results. As shown in the result, the proposed checkpoint procedure slightly affects the overall performance because the latch is not used to perform the checkpoint at all. The overall performance is degraded from 1% in one user to 5% in eight users during checkpoint time. The overall slight performance degradation comes from the CPU time for threads and OS costs for resource scheduling and other actions.

![Figure 5.10 Effect of Checkpoint on Transaction Processing](image)

Another parameter of interest is the time it takes to recover a database after a system crash. Recovery time comprises of time to load the entire database and then apply the system log
residing on disk. The former time is fixed and the later time is proportional to log size. Figure 5.11 plots the time to recover the database (in second) against the size of the system log (in MB) applied during recovery. In the experiments, before performing restart recovery, the file system cache is flushed – thus, no log records or database pages are cached at the start of recovery processing.

From Figure 5.11, it follows that the total time to recovery as well as the recovery time increase somewhat linearly with the log size. The total time is sum of the loading time of database and recovery time. The recovery time is sum of read time of logs and CPU time consumed by recovery algorithm in processing log records. The fixed time to load the entire database (1GB) at the start of recovery is about 25 seconds. The recovery time takes 0.7 seconds to process every additional 10 MB of the system log.

![Figure 5.11 Recovery Times vs. Log Size](image)
5.5 Performance of Replication Mechanism

This section shows the performance of four kinds of replication models in the proposed replication scheme. The TPS values in stand-alone, master-slave, master-master, master-multi slave, and master-multi slave with propagator model are demonstrated in Figure 5.12. The stand-alone model obviously outperforms other replicated models. The performance of replicated model however shows the almost same results to one of stand-alone system. The TPS of the master-slave model is estimated as the 95% performance of stand-alone system. In the master-multi slave model, the replication overhead of master server may be higher than one of the basic master-slave model as the number of standby server increases. In the worst case, the TPS of the master-multi slave model including two slaves shows the 90% of TPS of the stand-alone model. The TPS of the master-multi slave with propagator model including two slaves, however, is degraded at most as 93% performance of the stand-alone system, since the propagator has the role of update propagation instead of master server. Actually, this model under the evaluation has three slaves including the propagator. Finally, the performance of the master-mater model including one-to-one is degraded at most as 89% performance of the stand-alone system, since each node updates itself and receives the updates of other relative master node. The overhead of replication in the proposed lazy-flex replication scheme can be ignorable as compared to acquiring the availability.
Figure 5.12 Throughputs for Various Replication Models

5.6 Overall Performance of Storage Management System

The proposed methods for transaction processing with high performance and availability are adopted on the storage manager of ALTIBASE. So, overall performance of the storage manager will be need analyzing. The representative result is in Table 5.2. It shows the TPS of single user environment. The uppermost TPS value was measured in the experiment in which transactions are completely read-only (select) transactions. This result is comparatively higher than other commercial products of MMDB.
Table 5.2 TPS of Single User Environment

<table>
<thead>
<tr>
<th>Types</th>
<th>Insert</th>
<th>Update</th>
<th>Read</th>
<th>Delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPS</td>
<td>6,772</td>
<td>5,534</td>
<td>7,816</td>
<td>6,383</td>
</tr>
</tbody>
</table>

Figure 5.13 demonstrates that TPS increases steadily with the number of users until the user requests exceed the capacity of CPU utilization at the HP platform. This experiment shows how the number of users affects the TPS. After about 8 users, the TPS is slightly degraded because of the limit of CPU utilization. The TPS of HP system shows the better performance than the other commercial systems even though it has the heavy load of transactions. The proposed method shows better performance than other commercial MMDBMS Timesten [TT] in case of update transaction especially.

![Figure 5.13 Scalability of the Storage Management System](image-url)
Figure 5.14 shows the test results of TPC-H benchmark [TH]. The 22 queries of TPC-H specification are run on the HP platform environment. Figure 5.15 also demonstrates the results of Wisconsin benchmark [DJT83]. The full set of standard benchmark test was performed since the query model of the MMDBMS implemented the proposed transaction processing techniques fully supports the standard SQL2 statements. The results are reasonably better than other MMDBMS.

Figure 5.16 shows the TPS of four types of transaction with respect to the data type. This experiment was performed with single transaction environment and a unique index. It also uses five built-in data type such as char, date, varchar, numeric, and integer. Upon the TPS results, the date type shows the worse performance than other built-in types and, on the other hand, the integer has the best result. The other types show the very similar performance.
Figure 5.15 Wisconsin Benchmark Results

Figure 5.16 Effects of Data Types on Performance
6 Conclusion

This thesis proposes the efficient transaction processing techniques of storage manager in MMDBMS for high performance and availability which can be applied into general database applications, such as soft switch systems, real-time billing systems, and customer authentication systems on the fields of internet services and mobile communication. The proposed methods provide the enhanced transaction processing methods for the high performance and fault tolerant database applications requiring predictable response time, high concurrent users, and high availability.

The proposed transaction processing techniques, specially, are four efficient methods for performance, stability, and availability of main memory database. First, the MVSL concurrency control method is the combined the versioning and the locking mechanisms, which has minimal latching to data, simple locking for record, and fewer lock information management. It has higher performance than conventional MVCC and 2PC which are 10% and 60% respectively on eight concurrent user environment. MVSL, specially, has good performance in which are many read transactions, because it used to multi versioning and no latching concurrency control for read transactions.

Secondly, PML-tree is that includes partial keys, a maximum key and a next link pointer in an index node and provides tree operations considering cache coherence under multiprocessor computers. The read and write operations of PML-tree are faster 10% and 30% respectively than the T-tree with index node versioning on sixteen concurrent users. These results are leaded from that PML-tree used to latch free tree traversal, logical version index concurrency control, and cache conscious algorithm.

Thirdly, multilevel logging and recovery methods are suggested to complement of trade-off of transaction durability and performance scalability. The multilevel logging has
four logging levels and since it has no analysis phase, the recovery method fast executes the recovery phase. Especially, latch-free fuzzy checkpoint is a fuzzy checkpoint that has not any latch on the dirty pages on checkpointing time and writes the dirty pages into two database files in rotation. The overall performance is degraded at most 5% during checkpoint time in many concurrent users.

Lastly, the lazy-flex database replication method is that allows any node to update any local data and to propagate the updates to the replicas at the destination nodes without global coordination. This method can configure various replication models flexibly, such as master-slave, master-master, and master-multi slave. The lazy-flex replication uses the log-based replication since the replication performance could be fast during the replication phase. Also, this method provides the various data synchronization protocols corresponding to the applied replication models. The performances of the various replication models on lazy-flex replication scheme are slightly degraded to compare stand-alone system, such as master-slave, master-multi slave, and master-master 95%, 90%, and 89% of performance of stand-alone system respectively.

This thesis also presents how the proposed methods are improved upon conventional methods and how they are implemented into the storage manager of the practical commercial main memory database system ALTIBASE. The implemented system shows better performance than other commercial MMDBMS. The system has been applied to the fields of commercial internet service, critical financial service and communication system equipments.

Further researches related on this thesis are to analyze the possibly existed problems of the proposed methods. In order to overcome the restricted database size or physical main memory size, on the other hand, there are two topics for further research. The one is to research the storage management model with supporting multi-level storage such as
memory and disk. The other is small sized storage management model for mobile equipments or diskless platforms such as PDA and telematics terminals.
References


[BPR’96] Philip Bohannon, James Parker, Rajeev Rastogi, S. Seshadri, Abraham Silberschatz, and S. Sudarshan, “Distributed multi-level recovery in


감사의 글

석사 과정 때 인생의 지표가 되시고 지금도 변치 않는 연구 열정과 한없는 사랑을 베풀어 주시는 배경영 지도 교수님께 존경과 감사의 마음을 드립니다. 그리고 기꺼이 섬사 위원장을 맡아주신 유형선 교수님, 섬사 위원이신 유상봉 교수님, 그리고 컴퓨터정보공학과 교수님들께 감사를 드립니다.

연구에 바쁜 가운데도 논문 섬사를 맡아주시어, 논문 섬사 과정에서 부족한 점을 지적하여 주시고 격려해 주신 한국전자통신연구원의 김명준 박사님과 한국과학기술원의 이윤준 교수님께 감사를 드립니다.

그리고 학위 과정 동안 많은 일을 도와준 데이터베이스연구실 후배들, 특히 박순영, 김영근 학생에게 고마운 마음을 전합니다. 또한 박사 과정 동안에 많은 조언과 응원을 보내 주신 역기숙 교수님, 구홍서 교수님, 이순조 교수님, 김재홍 교수님, 이재동 교수님, 심종익 교수님, 김영배 박사, 이진수 선배님, 그리고 논문 작성에 도움을 준 이규웅 박사, 후배이자 직장 동료인 대일이와 환재, 알티베이스의 모든 분께 감사의 뜻을 전합니다.

오늘이 있기에 깜짝 대접을 잘해 주시고 항상 겸손하신 형님 형수님들, 누나와 배형, 그리고 아들처럼 사랑해 주시고 우리 아이들을 예쁘고 건강하게 키워주신 장인 장모님, 처가 직구들 모두에게 심심한 감사를 드립니다.

끝으로 여름과 힘든 가운데에서도 믿고 큰 힘이 되어준 사랑하는 아내와 아직도 어리게만 보이지 않은 돈을 벌어 불우한 노인들을 돕겠다는 중학생 아들 도훈 그리고 장례에 와 의사가 되어 엄마 아빠를 늘지 않게 해주겠다는 말 예원이와 함께 이 기쁨을 함께 나누고자 합니다.

2004년 12월 29일