Implementing Storage Manager in Main Memory DBMS ALTIBASE^{TM} *

Kwang-Chul Jung[1] and Kyu-Woong Lee[2] and Hae-Young Bae[3]

[1] Real-Time Tech. Lab.
ALTIBASE Co. Seoul, KOREA
jungkc@altibase.com

[2] Kyu-Woong Lee
Dept. of Computer Science
Sangji Univ. Wonju, KOREA
leekw@sangji.ac.kr

[3] Hae-Young Bae
School of Computer Engineering
Inha Univ. Incheon, KOREA
hybae@inha.ac.kr

Abstract. We present design and implementation techniques for storage management in main memory database systems. ALTIBASE^{TM} is the relational main memory DBMS that enables us to develop the high performance and fault tolerant applications requiring predictable response time and high availability. It provides the short and predictable execution time and the storage for the various type of enormous data as well as the basic functionality of disk resident general DBMS. In this paper, we give an overview of ALTIBASE^{TM} system architecture and describe the implementation techniques and principal data structures. Especially, the particulars in design of storage manager in ALTIBASE^{TM} are precisely illustrated and major data structures for the storage management are explained.

1 Introduction

General-purpose disk resident database systems failed to meet the needs of applications requiring short, 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. Current DBMS applications such as call routing and switching of telecommunications and mobile applications, however, require

* This work was done as a part of Information & Communication Fundamental Technology Research Program supported by Ministry of Information & Communication in republic of KOREA.

the fast response of transactions, the persistency of data, and storage for the vast multimedia data.

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. 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. When the entire database resides in the main memory, implementation techniques developed under the assumption of disk I/O as the main cost of database operations should be reexamined. Hence, we present the design and implement techniques for the main memory database ALTIBASE^{TM} which is adequate to the application that requires the high performance.

In this paper, we give an overview of our system architecture and related design and implementation techniques for main memory database systems. The remainder of paper is organized as follows. In section 2, we provide our system architecture and functional model. Section 3 explains the implementation techniques and principal data structures of storage manager in ALTIBASE^{TM}. The physical structures for memory data representation are depicted and memory, transaction, index, and recovery management methods are illustrated. In section 4, we evaluate the transaction performance of ALTIBASE^{TM} system under various circumstances and we give our conclusions in section 5

2 Overview of ALTIBASE^{TM} Architecture

ALTIBASE^{TM} system consists of five major components: interface, query processing manager, session manager, replication manager, and storage manager as shown in Figure 1. ALTIBASE^{TM} system provides the various industrial standard API's such as ODBC, JDBC, SQL CLI(Call Level Interface), and ESQL programming interface. The administrative interface and monitoring tools for DBA is also available to control the system, and end user can use the standard SQL statement directly using the *iSQL* tool. The communication layer and session manager is responsible to manage the network connection between database clients and server system. When the client application starts, a session is created and it is maintained by session manager until the client is normally terminated or aborted. ALTIBASE^{TM} system perfectly supports the standard

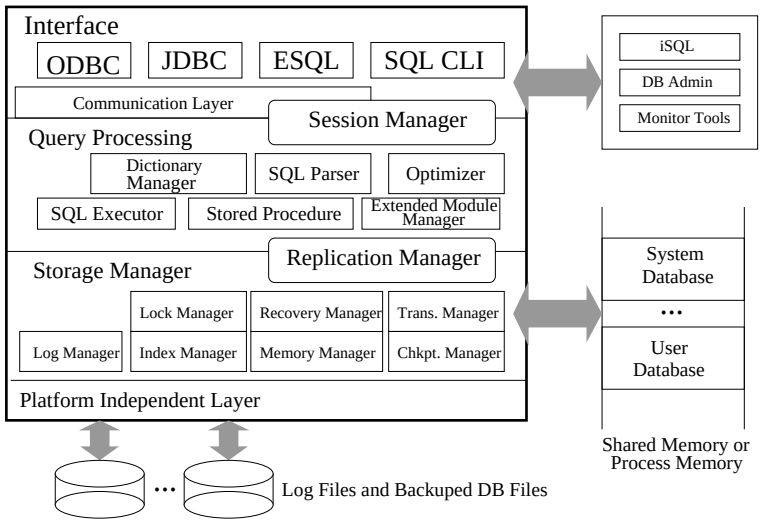

Fig. 1. Overall Architecture of ALTIBASE^{TM}

SQL2 statements, even though the most commercial MMDBMS does not have the capability to process the full set of SQL statements. The query processor of ALTIBASE^{TM} supports query optimization, nested join, sort-merge join and join with hash and sort operation. In order to support these capabilities, the query processor consists of data dictionary manager, SQL executor, optimizer, SQL parser and stored procedure. The storage manager is the core part of the server process that supports main capabilities of DBMS. It consists of the memory, transaction, index, recovery, log, and lock manager [JLB03]. The design and implementation techniques of those sub-components are illustrated in section 3. In order to acquire the higher concurrency and more efficiency, our transaction manager uses the multiversion concurrency control mechanism and adopts the multiple granularity locking [BC92, AK91]. The application that consists of almost read operations take a great advantage of being executed concurrently in our system. The recovery manager guarantees the ACID property of transaction perfectly. It hence have to perform the synchronization operation between memory and disk storage for the transaction durability [JSS93, GMS93]. However, it causes the substantial bottleneck of overall performance, since the synchronization needs to write the log records into physical disk storage. Our recovery manager suggests the *memory mapped file* or *memory buffer* mechanism as the log buffer in order to eliminate the unnecessary synchronization operation with disk. In our ALTIBASE^{TM}, transaction durability furthermore can be adjusted to the various levels corresponding to the significance of data. Depending on these durability levels, the certain types of log records are not necessary to be maintained and the smaller set of logs have to be preserved. The replication

manager is placed between query processor and storage manager, since our basic replication mechanism is performed based on the update log and its transactional execution. Our 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.

3 Implementation Techniques for Storage Manager

As shown in Figure 1, the storage manager is the core essential component in the ALTIBASE^{TM}. In this section, we first depict physical structures of records and pages and then describe design issues of each component in our storage manager such as the memory, index, transaction and recovery manager.

3.1 physical structures of the record and page

A record is the most fundamental data structure for storing the user and meta data. In our system, there are two kinds of records, fixed and variable length records. The fixed length record includes the data of fixed length column such as numeric value or fixed length character and the link value to the variable length column. The variable length record is the storage unit for variable length column such as varying size characters and multimedia data. Figure 2 depicts the fixed and variable length record. The record header of fixed length record has the structure type *smcSlotHeader* and it includes the information on a transaction sequence number, transaction identifier, a link to the next version record and another data for record. The *m_scn* is the system commit number that can be incremented by one when the transaction will be committed successfully and it should be limited as maximum value 2^{62}. In our ALTIBASE^{TM} system, the multiversion concurrency control method is adopted to control transactions that are concurrently executed. Hence, the *m_scn* is used as the version number of data age. When a transaction reads the certain data, the proper version data can be found by using this *m_scn* value. The *m_tid* is the transaction identifier which is the unique value during the system runtime. The *m_next* is a link value for the next version object identifier. When this record has been updated or deleted, *m_next* value is used to find the proper version of record. The *m_used_flag* and *m_drop_flag* indicates whether this record is allocated and deleted respectively. The header of variable length record has the type *smcVarColumn* which can be linked by a pointer value of fixed length record. The structure *smcVarColomn* has a object identifier, and its length, and other flags as show in Figure 2. The *OID* member variable is the pointer value to indicate the next variable length record and the *length* has different meanings whether this struct variable is used in a fixed length record or in a variable length record. The *length* value means the total length of the contained variable length record when it used in the fixed length record. In the case of variable length record, however, the *length* value denotes the length of its own record.

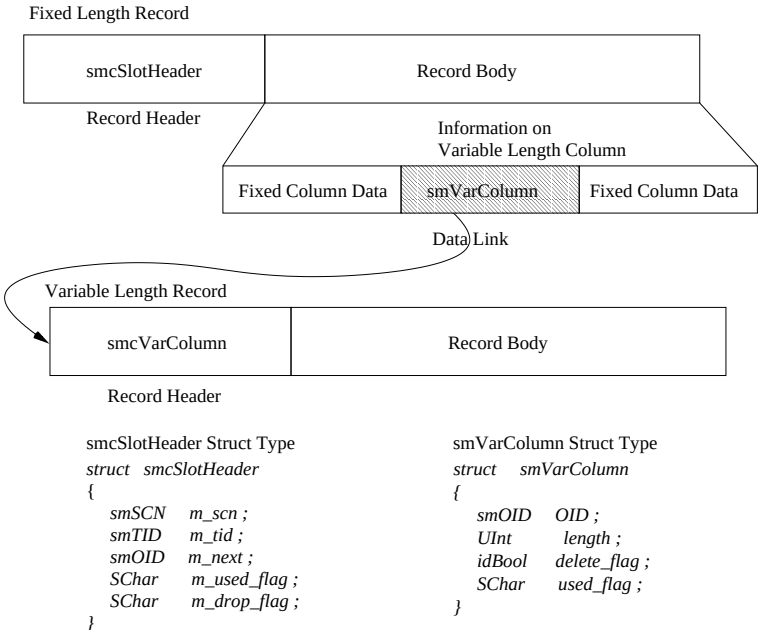

Fig. 2. The Fixed and Variable Length Record

A page consists of the page header and body. The page header has the information on each own page and the body is constructed by a set of records. A page can be classified as the temporary and persistent page whether the page is in the memory pool or occupied currently. The temporary page is basically the allocated space from available memory pool and the persistent page is actual data page on the memory that should be written into disk. The layout of temporary and persistent pages are depicted in Figure 3. The page header of temporary page has a set of pointer values $< m\_self, m\_prev, m\_next >$ that denotes its location of page in the memory space. The temporary page does not have the page identifier because it is not allocated yet and simply in the memory pool. The header of persistent page consists of a set of values $< m\_self, m\_type, m\_prev, m\_next >$. The *m_self* indicates the its own page identifier and the *m_prev* and *m_next* points to the previous and the next page identifier, respectively. Upon the records in a page, the persistent pages are classified as two groups such as the fixed and variable page. The fixed page has only fixed length records and the variable record, similarly, contains only variable length records. The *m_type* denotes page type and has one of value in the enumerated type *SmcPageType* as shown in the Figure 3. There are several kinds of variable page corresponding to the size of variable length records that will be included. The variable length record with the size from 32 bytes to 8K bytes should be stored into a proper size of variable

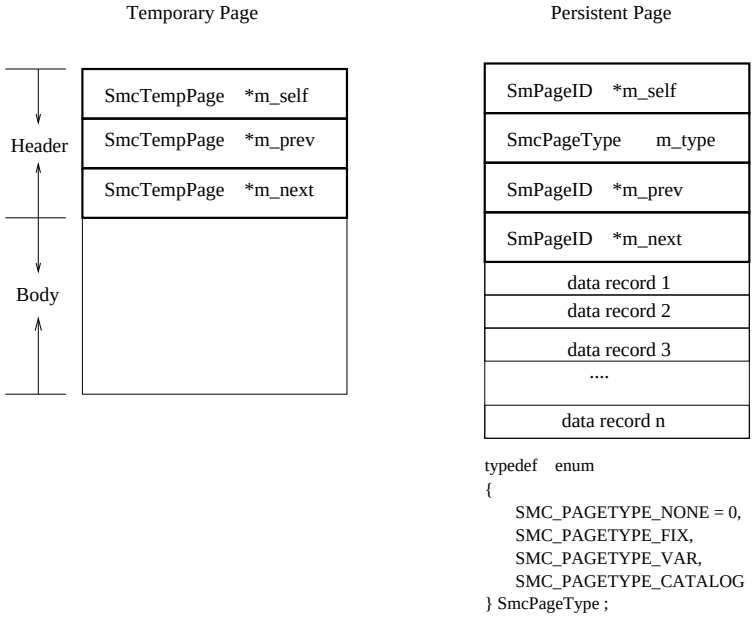

Fig. 3. Temporary and Persistent Pages

page. Hence, the variable page should have the additional member field, named *m_idx*, in the page header which indicates the maximum length of the variable record that can be stored in its variable page.

Pages also can be categorized into the catalog and normal pages by the records that are included in their pages. The catalog page contain the meta data to maintain the entire consistent database state. In this case, as stated above, the page type *m_type* should be *SMC_PAGETYPE_CATALOG*. There are only one catalog page per a database and it is constructed with the page identifier 0 at the time of the database creation. The catalog page is loaded firstly into memory when the database system starts and then the entire database pages are loaded based on the meta data in the catalog page. Figure 4 shows an instance of catalog page as an example. The catalog page has the information on a database such as the number of used data pages and available pages, the next page identifier. We define the type *smmMembase* to manage those information as illustrated in Figure 5. The *smmMembase* consists of two areas, the database information area and the data page control area. The database information area are filled up from the control file of ALTIBASE^{TM} when the system starts. It contains the general information about the database such as the database name, the size of log and so on. The *m_alloc_pers_page_count* of data page control area denotes the number of data pages that are allocated currently and *m_free_pers_page_count* denotes the number of free data pages. The *m_first_free_pers_page* indicates an object

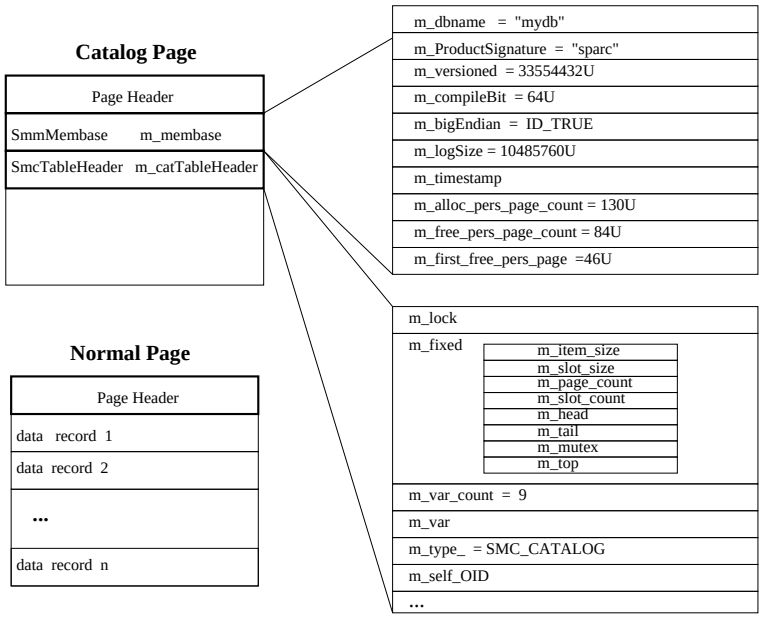

Fig. 4. An Example of Catalog and Normal Pages

identifier of the data page that will be allocated at the next time. The catalog page also has the additional information on the table header as well as the basic page header. The type *smcTableHeader* is used to maintain these table header information that includes the lock information on the table, the pointer to the first fixed and variable data page, the type of table, the object identifier of the table, the synchronization flag, the columns information, and all other meta data to manage the table. When a transaction access a table, these information on the catalog table should be referenced and modified firstly in order to maintain the consistent database state.

3.2 Memory Manager

In main memory database systems, all data pages should be loaded into the main memory from disk when the system starts and these data pages are managed and controlled by the MMDBMS [WZ94]. The memory manager, *smmManager*, has the responsibility to manage these loaded data pages in the memory in ALTIBASE^{TM} system. The page control header(PCH) that is linked by *smmManager* is the principal structure for management of data pages in a memory. There is one page control header as a structure type *smmPCH* per a page and it contains the information about the page management such as the location of the page in the memory and the dirty page flag. The *PCH* memory block has

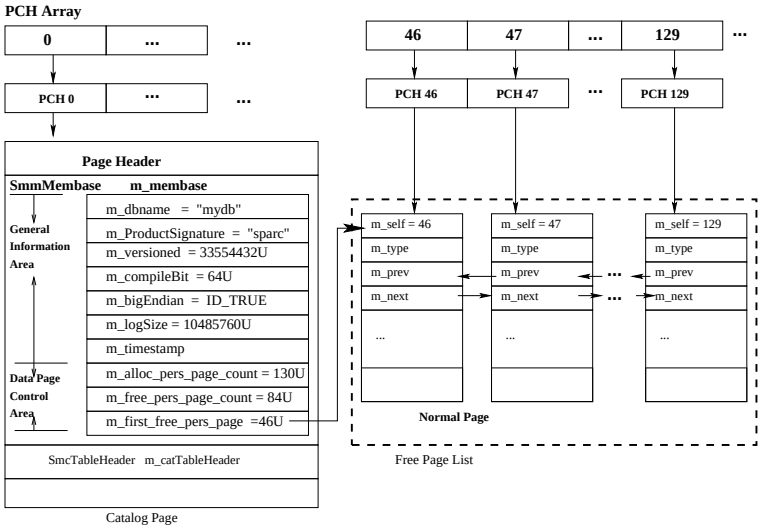

Fig. 5. An Example Structure *smmMembase* in the Catalog Page

to be allocated before the corresponding page is loaded into memory. When the page is loaded, the allocated *PCH* block should be mapped into the loaded data page as described in Figure 6. The memory manager *smmManager* have to keep up the control data of entire memory during the runtime of system. Figure 6 shows the description of our memory manager *smmManager* and its memory pool. The primary data that should be managed by *smmManager* is classified as four groups.

- The available memory pool for the actual data page and *PCH* memory block is managed. When more pages are required as a increase of database, the page should be fetched from the page memory pool and it is linked into the *free page list* as depicted in Figure 6. The *PCH* block is also obtained from *PCH memory pool* and it is mapped into a proper data page.
- The array of *PCH* is managed. When the system starts or the database is enlarged, *PCH* are allocated and is linked into the proper page. The each element of *PCH* array points to the *PCH* block as shown in Figure 6.
- The general information about a database is maintained by using the structure *smmMemebase* as illustrated in previous section 3.1.
- The catalog data of database is managed through the structure *smcTable-Header*.

Figure 7, especially, illustrates the relationship among the array of it PCH, the *PCH* block, and the persistent page. The array *m_PCHArray* is initialized with the number of elements as much as the number of data pages that can be supported by the system. The maximum number of elements should be given

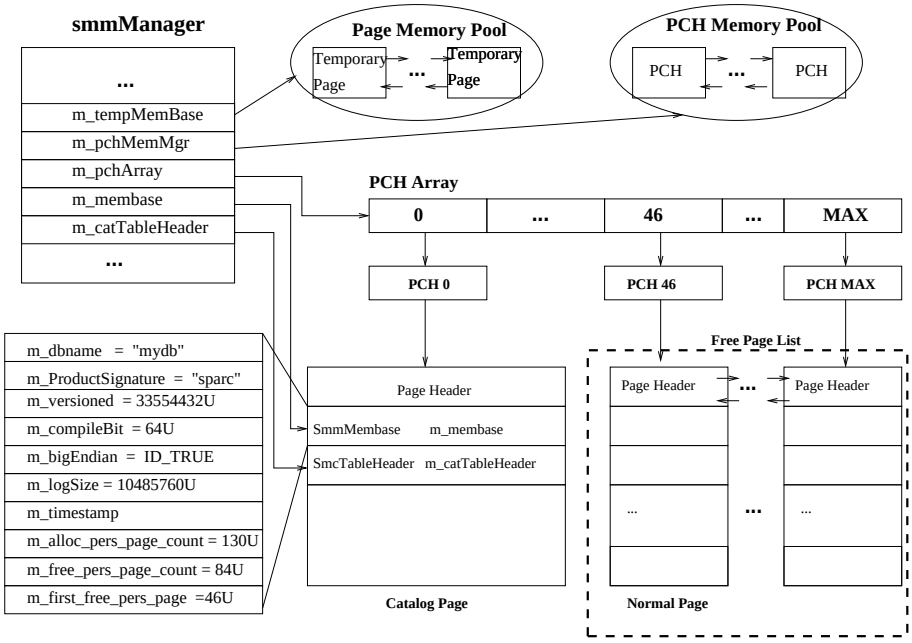

Fig. 6. Memory Manager *smmManager* in ALTIBASE^{TM}

as the system parameter in the control file. Each element of *PCHArray* has the pointer to one *PCH* block. The structure type *smmPCH* for one *PCH* block actually contains a pointer to the actual persistent page *m_page*, a *m_mutex* flag for mutual exclusive access, the dirty flag *m_dirty*, and the pointer to the next dirty page control header *m_pnxtDirtyPCH*. The *m_page* indicates the location of data page in the memory and it is necessitated during the actual data access. The *m_mutex* is the flag for concurrency control among transactions. The flag *m_dirty* and *m_pnxtDirtyPCH* is set when the corresponding data page is modified. The *m_pnxtDirtyPCH* is used to manage the list of modified data pages. This link value is extremely helpful to accelerate the performance of data access. Moreover, the index of *m_PCHArray* is the identical to the data page identifier in order to be achieved the high performance. For example, the data page 3 can be directly accessed through the third *PCH* element in the *m_PCHArray*.

The structure *smmPCH* is allocated from memory when the database is created or the system starts. It can be also allocated during the recovery or database expansion phase at online state. In these cases, an empty *PCH* block and a temporary page is fetched from the *PCH* memory pool and the page memory pool, respectively and then they are mapped into each other. An example scenario of data page and *PCH* allocation is illustrated in Figure 8. In this example, we show the allocation scenario of the data page with the page ID 10. An empty

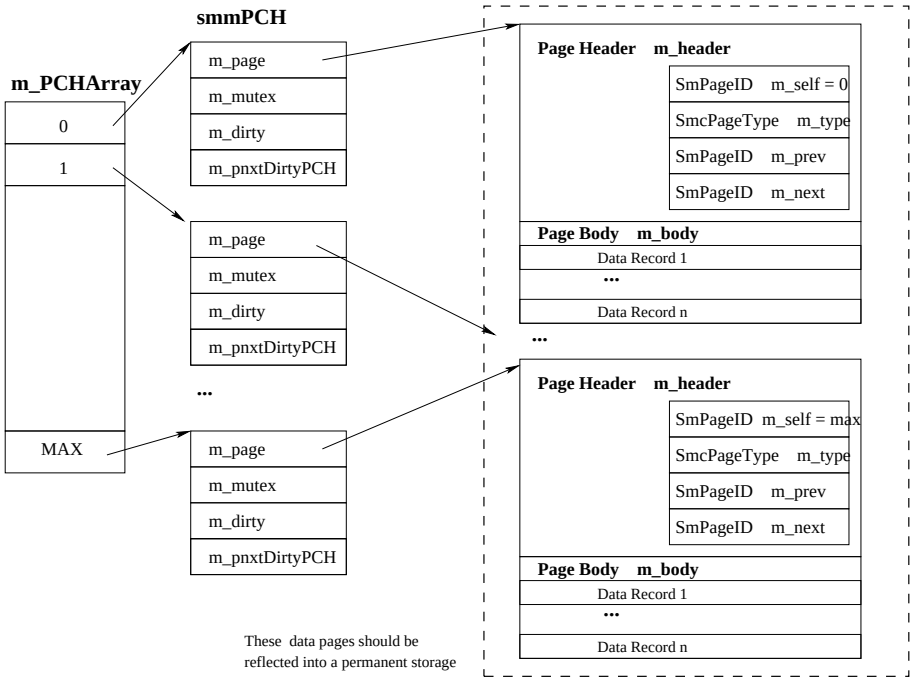

Fig. 7. The PCH Array *m_PCHArray*

PCH block is fetched from the *PCH* memory pool and then the index 10 of array *m_PCHArray* is linked to the fetched *PCH* block at step 2. A temporary data page is obtained from the page memory pool and it is transformed into the persistent data page at step 3. Finally, at step 4, the fetched *PCH* block is mapped into the persistent page and then we can access the data page with page ID 10 through the 10th *PCH* block of *PCH* array.

Table 1 describes the occasions when the *PCH* block is needed and the number of corresponding pages.

3.3 Index Manager

An index structure is the major factor that affects the overall system performance heavily. While the B-tree or B^+-tree is the most common index structure in disk-based database systems, the T-tree has been widely used as a index structure for main memory database systems.

The T-tree [LC92], rooted in the AVL tree and B-tree, is a balanced binary tree whose nodes contain more than one item. A node of a T-tree is called as a T-node that is depicted in Figure 9. A T-node consists of a number of data, 1 parent pointer and 0, 1, or 2 child pointers. An internal T-node has two child

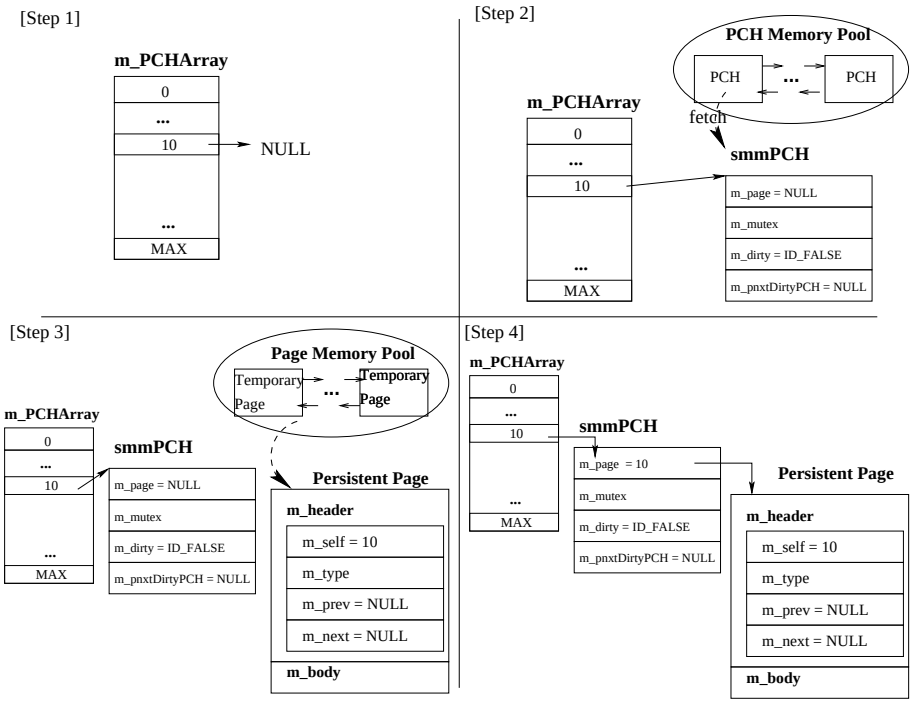

Fig. 8. An Example of Data Page and *PCH* Allocation

Table 1. *PCH* Allocation Time and The Number of Pages

DB Creation Time	the user-defined number of pages or at least 129
DB Restoring Time	the number of pages that exist in the database currently
Recovery Phase	the same as the number of pages that should be recovered
DB Expansion Phase	129

pointers pointing to its left and right subtrees, respectively. A leaf T-node has no child pointers and a *half-leaf* node has only one child pointer. The data pointers in a T-node point to the corresponding data entries in the main memory, hence through the data pointers, the corresponding data entries and and their keys can be accessed. A balance factor is a special data field in a T-node which is the value of the right subtree height minus the left subtree height. There are also two special fields *min* and *max* in each T-node that stores the minimum and

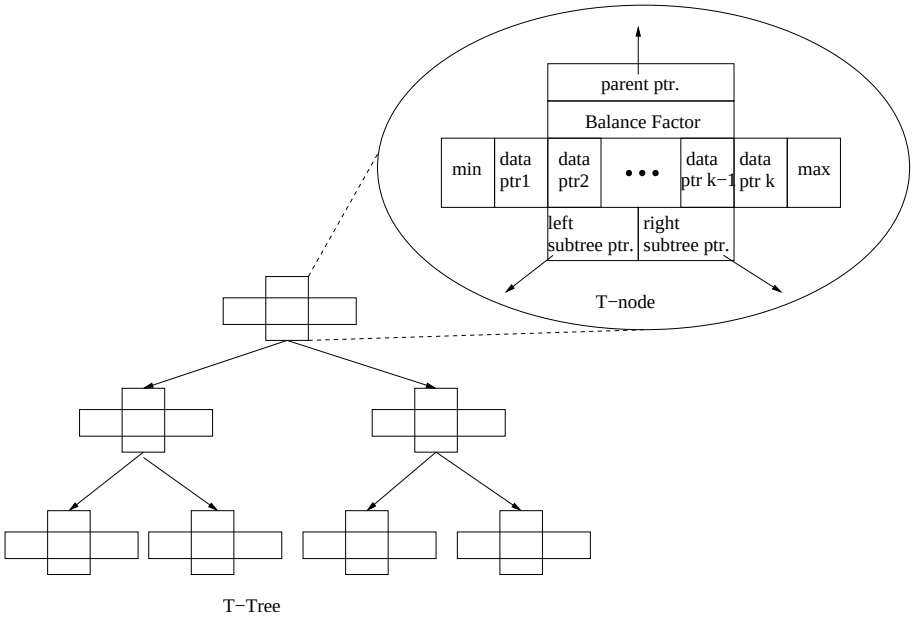

Fig. 9. Structure of a T-node in a T-tree

maximum key values in that node, respectively.

The T-tree index structure was adopted by several commercial MMDB 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 us to adopt a 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, in ALTIBASE^{TM} index manager, 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 a 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. We can traverse the T-tree more efficiently through the physical versioning

of a T-node upon the change operation on the T-tree.

Figure 10 shows our data structure for implementing the T-tree with a physical versioning. There is one structure *smntHeader* for a T-tree and there are

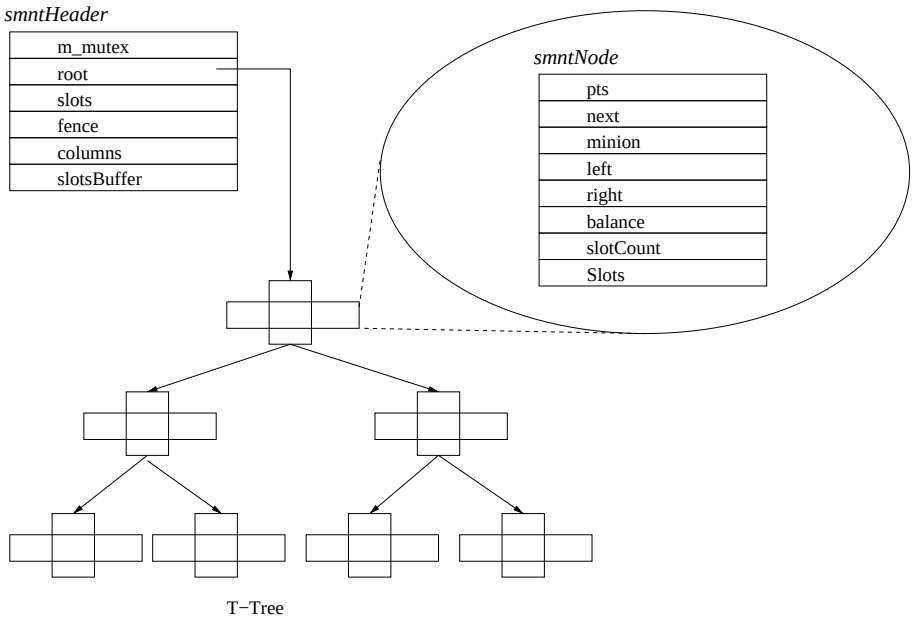

Fig. 10. Date Structure of for the Index Management

a number of structure it smntNode for T-nodes. The structure *smntHeader* has a *m_mutex* value for the concurrency control of T-tree. A *root* variable points a root node of a T-tree. The *SlotBuffer* value in the structure *smntHeader* is a pointer to a available slot list. A slot contains a data pointer to the corresponding data entry in the memory. The *slots* is a pointer to the current occupied slot list. The *pts* in the structure *smntNode* is a physical timestamp for multiversioning a T-node. The *next* values is used to manage the used list and free list for a slot and *minion* value is for the node that has the same version. The *left* and *right* is a pointer to the left and right subtree, respectively. The *Balance* values means the balance factor of the T-tree and the *slotCount* denotes the number of key value in the T-node currently.

3.4 Transaction Manager

The performance improvements and concurrency degree can be obtained for transaction management by employing the modified multi-version concurrency

control method in $ALTIBASE^{TM}$. In our transaction manager, we eliminate the unpredictable waiting for a data item that is a main disadvantage of conventional locking mechanisms. Transaction manager of $ALTIBASE^{TM}$ allows the data resource to be of various sizes and defines a hierarchy of data granularities. The small granularity of data can be controlled by multi-version concurrency control while the large granularity of data 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.

Our transaction manager supports the three types of isolation level, *consistent*, *repeatable*, and *no_phantom*. The *no_phantom* level is the same as the serializability of conventional concurrency control [RC96]. Each transaction sees a common serial order of update transactions that is consistent with their commit order. In the second isolation level *repeatable*, each transaction sees a serial order of update transactions and it is not necessarily the same as other executing transactions. This isolation level supports the weak consistency. The *consistent* isolation level is the same as the update consistency at the multi-version concurrency control [BC92, AK91]. All the data resources fetched by the current transaction are guaranteed that those are written by committed transaction completely in this isolation level. The data item written by uncommitted transaction cannot be fetched by any other transactions. Hence our transaction manager provides the different type of lock on the user table as illustrated in Table 2.

Table 2. Lock Types based on Isolation Level

Isolation / Operation	*Consistent*	*Repeatable*	*No_Phantom*
Read	IS	S	S
Write	IX	IX	X

IS : Intention Shared Lock IX : Intention Exclusive Lock
S : Shared Lock X : Exclusive Lock

The entire information on the transactions which are currently executed must be written on the *transaction table* and controlled by the transaction manager. When a transaction starts, an entry for a new transaction must be allocated from *transaction table* and it has to be returned when the transaction has been successfully committed. In this case, the *transaction table* should be intensively accessed since every transaction has to reference it. Our transaction manager therefore manages several *transaction free lists* in order to get a enhancement in the concurrent execution of transactions. The transaction manager hence has the responsibility for maintaining the *transaction table* and *transaction free lists*. We define the structure type *smxTransMgr* for the transaction manager to keep

up the information on the transaction management as depicted in Figure 11. The structure type *smxTransMgr* has two link values, named *m_arrTrans* and *m_arrTransFreeList* to *transaction table* and *transaction free lists*, respectively. The member variable *m_cTrans* denotes the number of the *transaction table* entry which is given by system parameter as the initial value 1,024. It means the number of concurrent transactions. The *m_cTransFreeList* means the number of *transaction free list*. This value is dependent on the number of CPU and cannot be changed during the system running time. For example, when the system has two CPUs, there are 8 *transaction free lists*. The *m_curAllocTransFreeList* represents which list should be currently used among *the given transaction free lists*. In order to control the concurrency, the *m_mustex* flag is used.

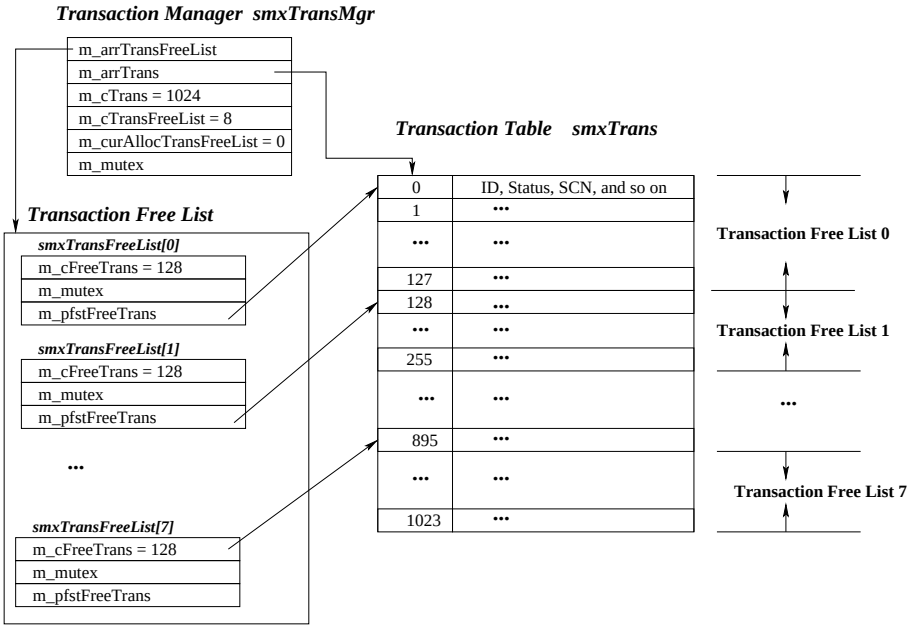

Fig. 11. Transaction Manager *smxTransMgr* in ALTIBASE^{TM}

The array of structure type *smxTrans* is used to keep up the *transaction table*. Each element is for the transaction that is currently executed. The *smxTrans* has the transaction identifier, the transaction status, the system commit number, and the link value to next transaction free list. The array of structure type *smxTransFreeList* is for the *transaction free lists*.

3.5 Recovery Manager

The basic discipline of recovery management is the *WAL(Write-Ahead Log)* method in *ARIES* [MHL+92, Moh99] system. Our 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, ALTIBASE^{TM} 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. Our *log flush thread* has the responsibility for 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 our logging mechanism, two kinds of log buffer can be used by our 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. We provide the memory buffer as a log buffer for alternative solution. Two kinds of log buffer, memory buffer and memory mapped file, is supplied and user can determine which log buffer will be used based on the transaction durability.

Therefore, ALTIBASE^{TM} provides multiple level of transaction durability. There are five levels of transaction durability for transaction performance and reliability of database state. Based on the transaction durability levels, the memory buffer or memory mapped file can be used as a log buffer and the synchronization techniques of log records are differently exploited. In *durability 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 durability level. The memory buffer is also used in *durability level 2*, but the logs must be synchronized with the log file by *sync thread*. The durability of committed transaction cannot be guaranteed in the *durability level 2* because the transaction is declared as *commit state* before its commit log record is synchronized with the log file. The memory mapped file as a log buffer is used in the *durability level 3*. Both the memory buffer and memory mapped file are used as a log buffer in *durability level 4 and 5*. In both levels, the log records are written into memory buffer and they should be synchronized with the memory mapped file by the *sync thread*. The durability of committed transaction is ensured in *level 5*, but not in *level 4*. Because the transaction is declared as *commit state* before the log records are ensured to be written into memory mapped file in *durability level 4*, we cannot assure the durability of transaction updates. In other hands, in *durability level 5*, the consistent log file at disk level is the necessary condition

for commit operation. There is trade-off relationship between the transaction durability and performance. We hence provide the various durability levels of transaction as the system parameter which can be controlled by the database administrator.

ALTIBASE^{TM} also supports and provides multiple levels of logging. There are three logging levels based on the importance between transaction performance and consistency of data. Different logging strategies are provided for each logging level. For only the purpose of fast response, transactions can be executed without leaving any log record in the *logging level 0*. The log records only for DML(*update, insert, delete statement*) should be managed in the *logging level 1*. The effect of committed transaction after checkpoint cannot be guaranteed to be completely reflected into database file in the *logging level 1*. All kinds of log records are manipulated and the recovery of every failure is ensured in the *logging level 2*.

4 Experiments

This section illustrates our experimental results for transaction execution of our $ALTIBASE^{TM}$. We focus on the number of TPS(transactions per second) based on the various number of records and concurrent users. All experiments were performed on *Sun Enterprise 3500* platform with 4 CPUs and 4G bytes of memory. Our experimental environment is described in Table 3.

Table 3. Environment for Experiments

Platform	Sun Enterprise 3500
Operating System	Sun Solaris 2.5.8
Number of CPU	4(Sparc Chip 400MHz)
Memory Size	4G Bytes
Number of Records	10,000 − 500,000
Number of Concurrent Users	1 − 50

All experimental transactions use the *native stored procedure* interface of $ALTIBASE^{TM}$. Our experimental results, therefore, are not interferenced with the unnecessary performance factor such as network delay, since the database access requests from the *native stored procedure* is directly sent to the query processor through the inter-process communication facility. *Select, insert, update, and delete transaction* were evaluated for the measurement of TPS under the strict condition of transaction durability 4 and logging level 2. 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 our *select transaction*. *Delete transaction* executes the deletion for one tuple with the search condition on the indexed key attribute.

The representative result is in Table 4. 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. The higher performance is evaluated in the *delete transaction* experiment over other DML transaction experiments because delete operations can be completed if we finished up to change only the value of data structure in memory manger. This result shows the TPS measured by our experiment is comparatively higher than other commercial products of MMDB.

Table 4. TPS of Single User Environment

Insert	Update	Select	Delete
6,134.97	4,405.29	29,411.76	12,345.68

Unit : TPS(Transactions per Sec.)

In Figure 12, the number of transactions per second(TPS) incurred in each of the transaction as the number of records is varied is plotted under the single user environment. In this experiment, the TPS is not very sensitive to the number of records because the features of efficient memory and index management are used and the CPU utilization is uniform. Hence, the TPS keeps the stable value or is slightly degraded beyond the limit of CPU utilization.

The number of concurrent users, however, affects the the number of TPS directly as depicted in Figure 13. As the number of users increases, the TPS also increases correspondingly because the ability of CPU is still enough until the number of user is up to about 10. However, the TPS keeps the stable plain line or is slightly degraded beyond the limit of CPU utilization. We can find that the reasonable scalability is guaranteed by $ALTIBASE^{TM}$ even though the experiment is performed under the heavy load environment.

5 Conclusion

ALTIBASE^{TM} is the relational main memory DBMS that enables us to develop the high performance and fault tolerant applications requiring predictable response time and high availability. In this paper, we primarily focused on the

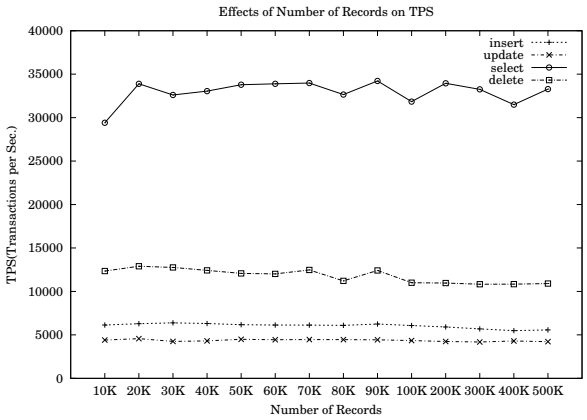

Fig. 12. Effects of Number of Records on TPS

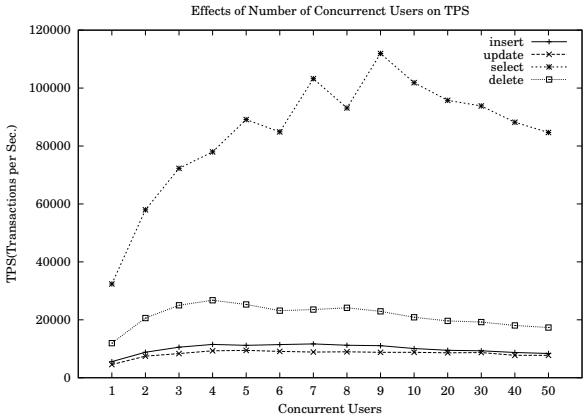

Fig. 13. Effects of Number of Concurrent Users on TPS

design and implementation of storage manager in our ALTIBASE^{TM} system. To gain the higher performance, we propose the fundamental database structure for memory management and flow control of memory manager. As the comparatively higher experimental results, $ALTIBASE^{TM}$ is ensured to guarantee the better performance in time-critical applications. The performance of ALTIBASE^{TM} also shows the reasonable results as compared to other commercial DBMSs. Currently, $ALTIBASE^{TM}$ version 3.0 is pronounced and is in the on-going project for hybrid function of disk-based facilities for large database.

References

[AK91] D. Agrawal and V. Krishnaswamy. "using multiversion data for non-interfering execution of wirte-only transactions ". In *Proc. of the ACM SIGMOD International Conference on Management of Data*, 1991.

[BC92] Paul M. Bober and Michael J. Carey. "multiversion qeury locking". In *Proc. of the 18th Conference on Very Large Database*, 1992.

[BPR+96] Philip Bohannon, James Parker, Rajeev Rastogi, S. Seshadri, Abraham Silberschatz, and S. Sudarshan. Distributed multi-level recovery in main-memory databases. In *Proc. of the International Conference on Parallel and Distributed Information Systems*, pages 44–55, 1996.

[GMS93] Hector Garcia-Molina and Kenneth Salem. "main memory database systems : An overview ". *IEEE Transactions on Knowledge and Data Engineering*, 4(6), 1993.

[JLB03] K-C. Jung, K-W. Lee, and H-Y. Bae. Design and implementation of storage manager in main memory database system altibase^{TM}. In *Proceedings of the 7thWorld Multiconference on Systemics, Cybernetics and Informatics*, 2003.

[JSS93] H. V. Jagadish, Avi Silberschatz, and S. Sudarshan. "recovering main-memory lapses ". In *Proc. of the 19th Conference on Very Large Databases*, 1993.

[LC92] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In *Proceedings of the 1992 International Conference on Very Large Database*, pages 294–303, 1992.

[LNT00] Hongjun Lu, Yuet Yeung Ng, and Zengping Tian. T-tree or b-tree: Main memory database index structure revisited. In *Australasian Database Conference*, pages 65–73, 2000.

[MHL+92] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems,*, 17(1), 1992.

[Moh99] C. Mohan. Repeating history beyond aries. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 1–17, 1999.

[RC96] Krithi Ramamritham and Panos K. Chrysanthis. "a taxonomy of correctness criteria in database applications ". *VLDB Journal*, 4(2), 1996.

[RSB+97] Rajeev Rastogi, S. Seshadri, Philip Bohannon, Dennis W. Leinbaugh, Abraham Silberschatz, and S. Sudarshan. Logical and physical versioning in main memory databases. *The VLDB Journal*, pages 86–95, 1997.

[WZ94] Michael Wu and Willy Zwaenepoel. envy: A non-volatile, main memory storage system. In *Proc. of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California*, pages 86–97, 1994.