IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661,p-ISSN: 2278-8727, Volume 17, Issue 2, Ver. 1 (Mar – Apr. 2015), PP 37-47
www.iosrjournals.org

Design & Development of an Advanced Database Management System Using Multiversion Concurrency Control Model for a Multiprogramming Environment

Mohammed Waheeduddin Hussain[1], Prof. P. Premchand[2],
Dr. G. Manoj Someswar[3]

[1]*Professor & Head, Department of CSE, Nawab Shah Alam Khan College of Engineering & Technology, Affiliated to JNTUH, Malakpet, Hyderabad – 500024, Telangana, India*
[2]*Professor in Department of Computer Science & Engineering, University College of Engineering, Osmania University. Hyderabad-500007, Telangana, India*
[3]*Professor & Dean (Research), Department of CSE, Nawab Shah Alam Khan College of Engineering & Technology, Affiliated to JNTUH, Malakpet, Hyderabad – 500024, Telangana, India*

Abstract: *Multi Version Concurrency Control (MVCC) is a locking scheme commonly used by modern database implementations to control fast, safe concurrent access to shared data. MVCC is designed to provide the following features for concurrent access: a. Readers that don't block writersb. Writers that fail fastMVCC achieves this by using data versioning and copying for concurrent writers. The theory is that readers continue reading shared state, while writers copy the shared state, increment a version id, and write that shared state back after verifying that the version is still valid (i.e., another concurrent writer has not changed this state first). This allows readers to continue reading while not preventing writers from writing and repeatable read semantics are maintained by allowing readers to read off the old version of the state.*

Keywords: *Lock based protocols, Time stamp based protocols, Two phase Locking, Deadlock Avoidance, Remote DataBackup, Log Based Recovery, Multi-core Systems*

I. Introduction

In a multiprogramming environment where more than one transactions can be concurrently executed, there exists a need of protocols to control the concurrency of transaction to ensure atomicity and isolation properties of transactions.

Concurrency control protocols, which ensure serializability of transactions are considered to be most desirable. Concurrency control protocols can be broadly divided into two categories:

1. Lock based protocols
2. Time stamp based protocols

Lock based protocols

Database systems, which are equipped with lock-based protocols, use mechanism by which any transaction cannot read or write data until it acquires appropriate lock on it first. Locks are of two kinds:

Binary Locks: a lock on data item can be in two states; it is either locked or unlocked.

Shared/exclusive: this type of locking mechanism differentiates lock based on their uses. If a lock is acquired on a data item to perform a write operation, it is exclusive lock. Because allowing more than one transactions to write on same data item would lead the database into an inconsistent state. Read locks are shared because no data value is being changed.[1]

There are four types lock protocols available:

Simplistic

Simplistic lock based protocols allow transaction to obtain lock on every object before 'write' operation is performed. As soon as 'write' has been done, transactions may unlock the data item.

Pre-claiming

In this protocol, a transactions evaluations its operations and creates a list of data items on which it needs locks. [2]Before starting the execution, transaction requests the system for all locks it needs beforehand. If all the locks are granted, the transaction executes and releases all the locks when all its operations are over. Else if all the locks are not granted, the transaction rolls back and waits until all locks are granted.

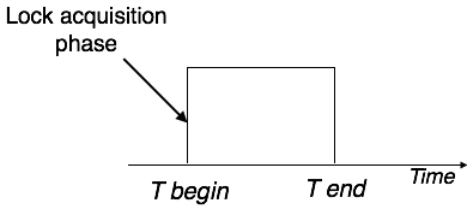

Figure 1: Pre-claiming

Two Phase Locking - 2PL

This locking protocol is divides transaction execution phase into three parts. In the first part, when transaction starts executing, transaction seeks grant for locks it needs as it executes. Second part is where the transaction acquires all locks and no other lock is required. [3]Transaction keeps executing its operation. As soon as the transaction releases its first lock, the third phase starts. In this phase a transaction cannot demand for any lock but only releases the acquired locks.

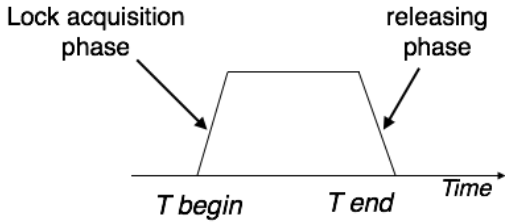

Figure 2: Two Phase Locking

Two phase locking has two phases, one is growing; where all locks are being acquired by transaction and second one is shrinking, where locks held by the transaction are being released.

To claim an exclusive (write) lock, a transaction must first acquire a shared (read) lock and then upgrade it to exclusive lock.

Strict Two Phase Locking

The first phase of Strict-2PL is same as 2PL. After acquiring all locks in the first phase, transaction continues to execute normally. But in contrast to 2PL, Strict-2PL does not release lock as soon as it is no more required, but it holds all locks until commit state arrives. Strict-2PL releases all locks at once at commit point.[4]

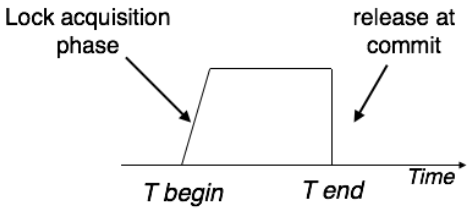

Figure 3: Strict Two Phase Locking

Strict-2PL does not have cascading abort as 2PL does.

Time stamp based protocols

The most commonly used concurrency protocol is time-stamp based protocol. This protocol uses either system time or logical counter to be used as a time-stamp.

Lock based protocols manage the order between conflicting pairs among transaction at the time of execution whereas time-stamp based protocols start working as soon as transaction is created.[5]

Every transaction has a time-stamp associated with it and the ordering is determined by the age of the transaction. A transaction created at 0002 clock time would be older than all other transaction, which come after it. For example, any transaction 'y' entering the system at 0004 is two seconds younger and priority may be given to the older one.

In addition, every data item is given the latest read and write-timestamp. This lets the system know, when was last read and write operation made on the data item.

Time-Stamp Ordering Protocol

The timestamp-ordering protocol ensures serializability among transaction in their conflicting read and write operations. This is the responsibility of the protocol system that the conflicting pair of tasks should be executed according to the timestamp values of the transactions.[6]

Time-stamp of Transaction Ti is denoted as $TS(T_i)$.

Read time-stamp of data-item X is denoted by R-timestamp(X).

Write time-stamp of data-item X is denoted by W-timestamp(X).

Timestamp ordering protocol works as follows:

If a transaction Ti issues read(X) operation:

If TS(Ti) < W-timestamp(X)

Operation rejected.

If TS(Ti) >= W-timestamp(X)

Operation executed.

All data-item Timestamps updated.

If a transaction Ti issues write(X) operation:

If TS(Ti) < R-timestamp(X)

Operation rejected.

If TS(Ti) < W-timestamp(X)

Operation rejected and Ti rolled back.

Otherwise, operation executed.

Thomas' Write Rule:

This rule states that in case of:

If TS(Ti) < W-timestamp(X)

Operation rejected and Ti rolled back. Timestamp ordering rules can be modified to make the schedule view serializable. Instead of making Ti rolled back, the 'write' operation itself is ignored.

In a multi-process system, deadlock is a situation, which arises in shared resource environment where a process indefinitely waits for a resource, which is held by some other process, which in turn waiting for a resource held by some other process.[7]

For example, assume a set of transactions $\{T_0, T_1, T_2, ...,T_n\}$. T_0 needs a resource X to complete its task. Resourc X is held by T_1 and T_1 is waiting for a resource Y, which is held by T_2. T_2 is waiting for resource Z, which is held by T_0. Thus, all processes wait for each other to release resources. In this situation, none of processes can finish their task. This situation is known as 'deadlock'.[8]

Deadlock is not a good phenomenon for a healthy system. To keep system deadlock free few methods can be used. In case the system is stuck because of deadlock, either the transactions involved in deadlock are rolled back and restarted.

II. Research Study

Deadlock Prevention

To prevent any deadlock situation in the system, the DBMS aggressively inspects all the operations which transactions are about to execute. DBMS inspects operations and analyze if they can create a deadlock situation. If it finds that a deadlock situation might occur then that transaction is never allowed to be executed.

There are deadlock prevention schemes, which uses time-stamp ordering mechanism of transactions in order to pre-decide a deadlock situation.

Wait-Die Scheme:

In this scheme, if a transaction request to lock a resource (data item), which is already held with conflicting lock by some other transaction, one of the two possibilities may occur:If $TS(T_i) < TS(T_j)$, that is T_i, which is requesting a conflicting lock, is older than T_j, T_i is allowed to wait until the data-item is available.If $TS(T_i) > TS(t_j)$, that is T_i is younger than T_j, T_i dies. T_i is restarted later with random delay but with same timestamp.[9]This scheme allows the older transaction to wait but kills the younger one.

Wound-Wait Scheme

In this scheme, if a transaction request to lock a resource (data item), which is already held with conflicting lock by some other transaction, one of the two possibilities may occur:

If $TS(T_i) < TS(T_j)$, that is T_i, which is requesting a conflicting lock, is older than T_j, T_i forces T_j to be rolled back, that is T_i wounds T_j. T_j is restarted later with random delay but with same timestamp.

If $TS(T_i) > TS(T_j)$, that is T_i is younger than T_j, T_i is forced to wait until the resource is available.

This scheme, allows the younger transaction to wait but when an older transaction request an item held by younger one, the older transaction forces the younger one to abort and release the item.[10]

In both cases, transaction, which enters late in the system, is aborted.

Deadlock Avoidance

Aborting a transaction is not always a practical approach. Instead deadlock avoidance mechanisms can be used to detect any deadlock situation in advance. Methods like "wait-for graph" are available but for the system where transactions are light in weight and have hold on fewer instances of resource. In a bulky system deadlock prevention techniques may work well.

Wait-For Graph

This is a simple method available to track if any deadlock situation may arise. For each transaction entering in the system, a node is created. When transaction T_i requests for a lock on item, say X, which is held by some other transaction T_j, a directed edge is created from T_i to T_j. If T_j releases item X, the edge between them is dropped and T_i locks the data item.[12]

The system maintains this wait-for graph for every transaction waiting for some data items held by others. System keeps checking if there's any cycle in the graph.

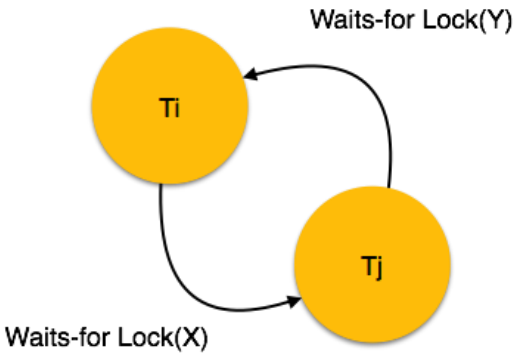

Figure 4: Wait-for Graph.

Two approaches can be used, first not to allow any request for an item, which is already locked by some other transaction. This is not always feasible and may cause starvation, where a transaction indefinitely waits for data item and can never acquire it. Second option is to roll back one of the transactions.

It is not feasible to always roll back the younger transaction, as it may be important than the older one. With help of some relative algorithm a transaction is chosen, which is to be aborted, this transaction is called victim and the process is known as ***victim selection.***

Failure with loss of Non-Volatile storage

What would happen if the non-volatile storage like RAM abruptly crashes? All transaction, which are being executed are kept in main memory. All active logs, disk buffers and related data is stored in non-volatile storage.When storage like RAM fails, it takes away all the logs and active copy of database. It makes recovery almost impossible as everything to help recover is also lost. Following techniques may be adopted in case of loss of non-volatile storage.[13]

A mechanism like checkpoint can be adopted which makes the entire content of database be saved periodically.State of active database in non-volatile memory can be dumped onto stable storage periodically, which may also contain logs and active transactions and buffer blocks.

<dump> can be marked on log file whenever the database contents are dumped from non-volatile memory to a stable one.

Recovery

When the system recovers from failure, it can restore the latest dump.

It can maintain redo-list and undo-list as in checkpoints.

It can recover the system by consulting undo-redo lists to restore the state of all transaction up to last checkpoint.

Database backup & recovery from catastrophic failure

So far we have not discovered any other planet in our solar system, which may have life on it, and our own earth is not that safe. In case of catastrophic failure like alien attack, the database administrator may still be forced to recover the database.

Remote backup, described next, is one of the solutions to save life. Alternatively, whole database backups can be taken on magnetic tapes and stored at a safer place. This backup can later be restored on a freshly installed database and bring it to the state at least at the point of backup.

Grown up databases are too large to be frequently backed-up. Instead, we are aware of techniques where we can restore a database by just looking at logs. So backup of logs at frequent rate is more feasible than the entire database. Database can be backed-up once a week and logs, being very small can be backed-up every day or as frequent as every hour.

Remote Backup

Remote backup provides a sense of security and safety in case the primary location where the database is located gets destroyed. Remote backup can be offline or real-time and online. In case it is offline it is maintained manually.

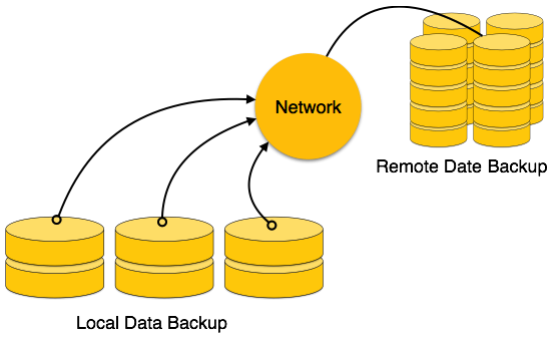

Figure 5: Remote Data Backup.

Online backup systems are more real-time and lifesavers for database administrators and investors. An online backup system is a mechanism where every bit of real-time data is backed-up simultaneously at two distant place. One of them is directly connected to system and other one is kept at remote place as backup.

As soon as the primary database storage fails, the backup system sense the failure and switch the user system to the remote storage. Sometimes this is so instant the users even can't realize a failure.

Crash Recovery

Though we are living in highly technologically advanced era where hundreds of satellite monitor the earth and at every second billions of people are connected through information technology, failure is expected but not every time acceptable.

DBMS is highly complex system with hundreds of transactions being executed every second. Availability of DBMS depends on its complex architecture and underlying hardware or system software. If it fails or crashes amid transactions being executed, it is expected that the system would follow some sort of algorithm or techniques to recover from crashes or failures.

Interpretation & Analysis

To see where the problem has occurred, we have generalized the failure into various categories as follows:

Transaction Failure

When a transaction is failed to execute or it reaches a point after which it cannot be completed successfully it has to abort. This is called transaction failure. Where only few transaction or process are hurt.

Reason for transaction failure could be:

Logical errors: where a transaction cannot complete because of it has some code error or any internal error condition

System errors: where the database system itself terminates an active transaction because DBMS is not able to execute it or it has to stop because of some system condition. For example, in case of deadlock or resource unavailability systems aborts an active transaction.

System Crash

There are problems, which are external to the system, which may cause the system to stop abruptly and cause the system to crash. For example interruption in power supply, failure of underlying hardware or software failure.

Examples may include operating system errors.

Disk Failure

In early days of technology evolution, it was a common problem where hard disk drives or storage drives used to fail frequently. Disk failures include formation of bad sectors, unreachability to the disk, disk head crash or any other failure, which destroys all or part of disk storage.

Storage Structure

In brief, the storage structure can be divided in various categories:

Volatile storage: As name suggests, this storage does not survive system crashes and mostly placed very closed to CPU by embedding them onto the chipset itself for examples: main memory, cache memory. They are fast but can store a small amount of information.

Nonvolatile storage: These memories are made to survive system crashes. They are huge in data storage capacity but slower in accessibility. Examples may include, hard disks, magnetic tapes, flash memory, non-volatile (battery backed up) RAM.

Recovery and Atomicity

When a system crashes, it many have several transactions being executed and various files opened for them to modifying data items. As we know that transactions are made of various operations which are atomic in nature. But according to ACID properties of DBMS, atomicity of transactions as a whole must be maintained, that is, either all operations are executed or none.

When DBMS recovers from a crash it should maintain the following:

It should check the states of all transactions, which were being executed.

A transaction may be in the middle of some operation; DBMS must ensure the atomicity of transaction in this case.

It should check whether the transaction can be completed now or needs to be rolled back.

No transactions would be allowed to left DBMS in inconsistent state.

There are two types of techniques, which can help DBMS in recovering as well as maintaining the atomicity of transaction:

Maintaining the logs of each transaction, and writing them onto some stable storage before actually modifying the database.

Maintaining shadow paging, where are the changes are done on a volatile memory and later the actual database is updated.

Log-Based Recovery

Log is a sequence of records, which maintains the records of actions performed by a transaction. It is important that the logs are written prior to actual modification and stored on a stable storage media, which is failsafe.

Log based recovery works as follows:

The log file is kept on stable storage media

When a transaction enters the system and starts execution, it writes a log about it

When the transaction modifies an item X, it write logs as follows:

$<T_n, X, V_1, V_2>$

It reads Tn has changed the value of X, from V1 to V2.

When transaction finishes, it logs:

$<T_n, commit>$

Database can be modified using two approaches:

1. **Deferred database modification:** All logs are written on to the stable storage and database is updated when transaction commits.
2. **Immediate database modification:** Each log follows an actual database modification. That is, database is modified immediately after every operation.

Design & Development

Multi-Version Concurrency Control

In our research work, we developed a multi-version concurrency control (MVCC) transaction manager. MVCC offers an alternative to the "pessimistic" database locking and can dramatically improve scalability and performance, especially in applications with one or more of the following characteristics: On-disk or hybrid (in-memory and on-disk) database storage

Many tasks or processes concurrently modifying the database (versus read-only)

Multi-core systems

Pessimistic database locking makes all or portions of the database unavailable to all but the single task that is updating it, thereby blocking other tasks. In practice, for an in-memory *eXtreme*DB database, this is often not an issue because transactions execute faster (in microseconds) than complex lock arbitration itself would take.

But even with the speed of the transactions, serializing a large number of concurrent tasks writing to the database can be a performance disadvantage. And even a few concurrent tasks writing to much slower file system-based tables in a hybrid database could be problematic for the MURSIW transaction manager (even the fastest solid-state disks, or SSDs, are more than an order of magnitude slower than RAM).

In contrast, MVCC is an optimistic model in which no task or thread is ever blocked by another because each is given its own copy (version) of objects in the database to work with during a transaction. When a transaction is committed, its copy of objects it has modified replaces what is in the database. Because no explicit locks are ever required, and no task is ever blocked by another task with locks, MVCC can provide significantly faster performance and greater utilization of multiple CPUs/cores.

Under MVCC, when tasks want to update the same data at the same time, a conflict does arise, and a retry will be required by one or more tasks. However, an occasional retry is far better, in performance terms, than the guaranteed complex lock arbitration, and blocking, caused by pessimistic locking. Further, in most systems, conflicts under MVCC are infrequent because of the logically separate duties among tasks--that is, task A tends to work with a different set of data than tasks B, C and D, etc.

Operation of MVCC

Figure 1 compares MVCC to pessimistic locking, in operation. The diagram shows three database tables, each with five rows, and three tasks that are reading and/or modifying certain rows of certain tables. Task 1 is modifying Table 1's row 3 (T1R3) and Table 2's row 5 (T2R5). Task 2 is modifying T3R1, T3R3 and T3R5. Task 3 is reading T3R3 and modifying T1R5 and T3R2. Note that there are two copies (versions) of T3R3: a copy in Task 2 and Task 3.

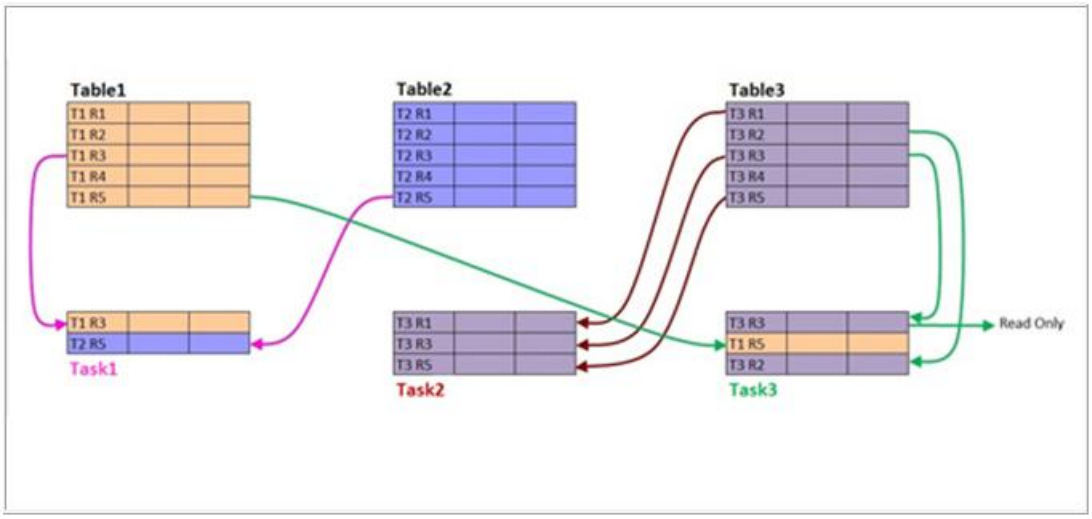

Figure 6: Comparison of MVCC with Pessimistic Locking

For purpose of this research work, we have assumed that all three tasks are started as close together in time as possible, but in the order Task 1, Task 2, Task 3. With MVCC, the three tasks run in parallel. With pessimistic locking, there are three possibilities: database locking, table locking and row locking. Each will exhibit different degrees of parallelism (but all less than MVCC).

Database locking: Task 1, Task 2 and Task 3 will be serialized. In other words, Task 1 will be granted access to the database while Task 2 and Task 3 are blocked as they "wait their turn." When Task 1 completes its transaction, Task 2 will run while Task 3 continues to wait. Finally, Task 3 will run after Task 2 completes its transaction.

Table locking: Task 1 and Task 2 will run in parallel because Task 1 acquires locks on Table 1 and Table 2, while Task 2 acquires locks only on Table 3. Task 3 will block until Task 1 and Task 2 complete because it also needs a lock on Table 1 (which is locked by Task 1) and Table 3 (which is locked by Task 2). Task 3 will be blocked for the length of time required by Task 1 or Task 2, whichever is greater.

Row locking: Again, Task 1 and Task 2 will run in parallel because they operate on different tables, (hence on different rows). Task 3 will again block because Task 2 has a write lock on T3R3, which Task 3 wants to read.

Performance Test for MVCC

Any serialization effectively defeats a multi-core system, because all but one core will be idle with respect to the utilization of the shared database. However, strategies to maximize parallelism, such as MVCC or fine-grained locking, impose their own overhead. In the case of fine-grained locking (row locking) there is lock arbitration, which can be complex. In the case of MVCC, there is version management--creating object versions, merging them and discarding them.

So for MVCC to be justified, the gain in parallelism has to outweigh the additional processing overhead. To illustrate, the graphs in Figures 2, 3 and 4 show the relative performance of in-memory database system on identical multithreaded tests executed on a multi-core system, using a multiple-reader, single-writer (MURSIW, or database-locking) transaction manager, and its multi-version concurrency control (MVCC) transaction manager.

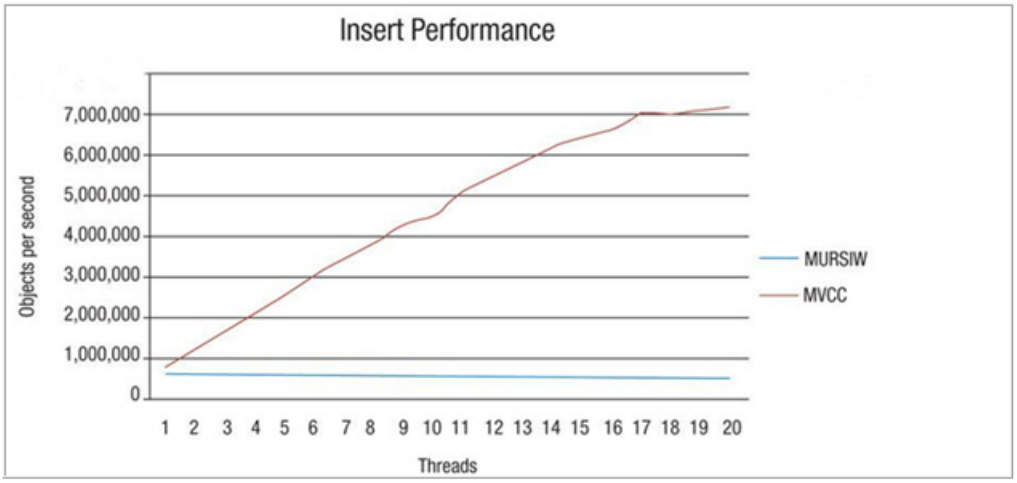

Figure 7: INSERT performance with MVCC (red) vs. MURSIW (blue) transaction managers

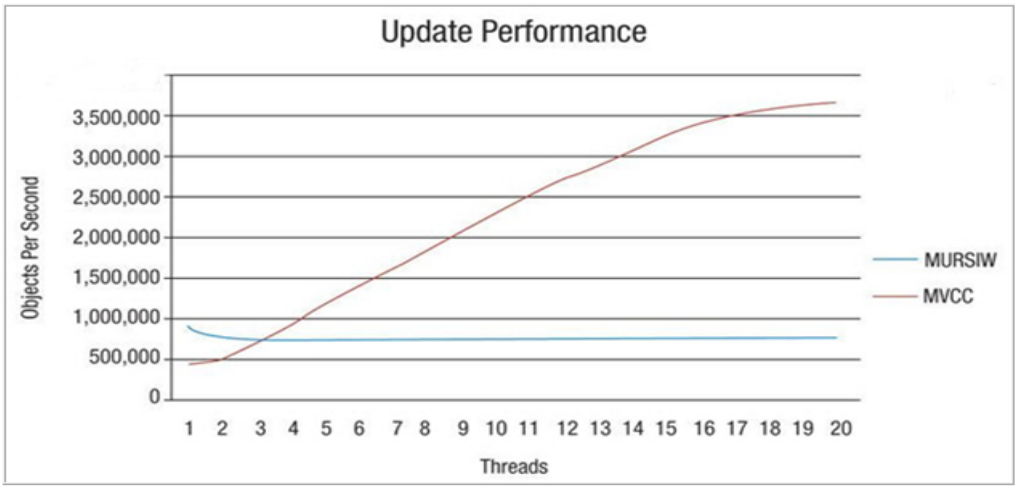

Figure 8: UPDATE performance with MVCC (red) vs. MURSIW (blue) transaction managers

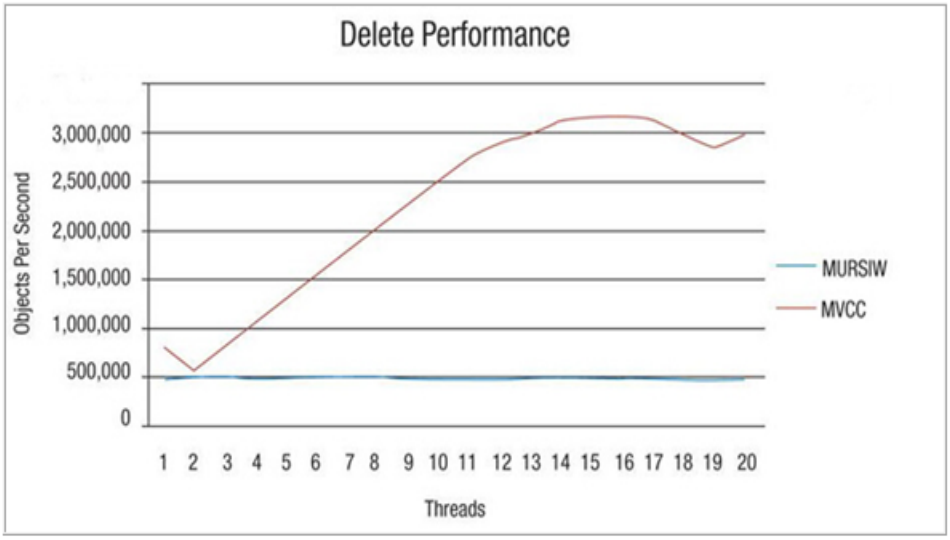

Figure 9: DELETE performance with MVCC (red) vs. MURSIW (blue) transaction managers

Note that the MURSIW transaction manager's serialization of read-write transactions (emphasis on the SIngle Writer characteristic) in the above example explains the nearly-flat performance line of MURSIW transactions as the number of cores increases from 1 to 20. In other words, if the MURSIW transaction manager can achieve 700,000 transactions per second, it can do so with one thread (1 thread executing 700,000 transactions-per-second) or with 20 threads (20 threads executing 35,000 transactions-per-second each).

Cursor Iteration, Hash & Tree Search

McObject also compared performance of MVCC and MURSIW transaction managers in performing three read-only operations: a cursor iteration using a database cursor to loop over every object in a table, as well as primary key searches using hash and tree indexes.

The results are very different from the write-intensive benchmark results, above. MURSIW will outperform MVCC for read-only operations because MURSIW is a very lightweight transaction manager. MVCC, on the other hand, has to make versions of objects, track them and discard them when they're no longer referenced in an active transaction.

The choice of MURSIW or MVCC should be a function of the ratio of query transactions to insert/update/delete transactions for a given application. Almost all applications are a blend. Because MURSIW is more efficient for database reads, applications in which reads predominate are may be better-served by the MURSIW transaction manager. Conversely, as the proportion of read-write transactions increases, so does the likelihood that MVCC will improve performance.

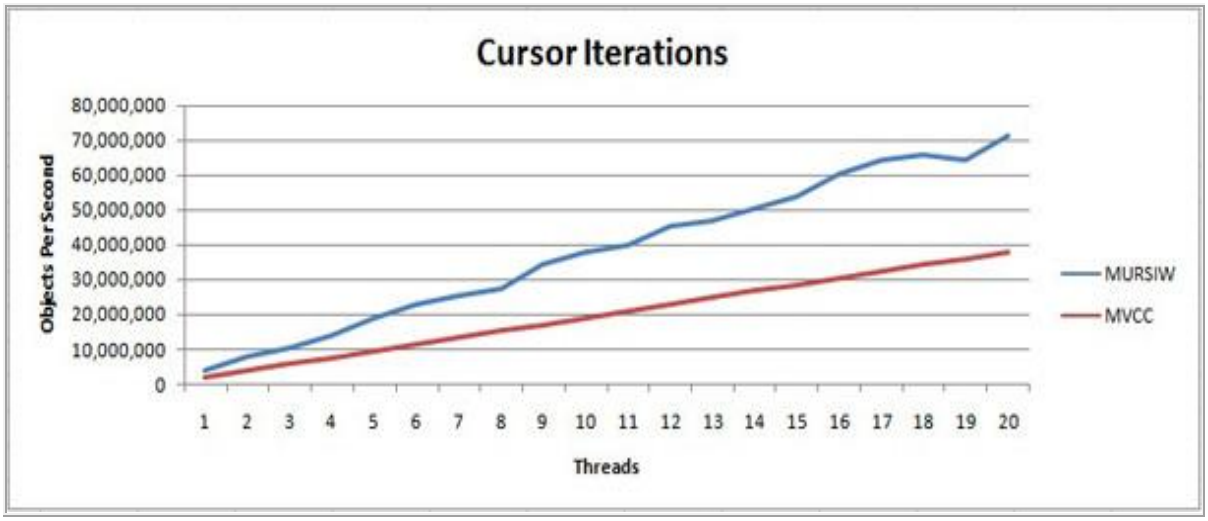

Figure 10: Cursor iteration, or database performance using a cursor to loop over every object in a table

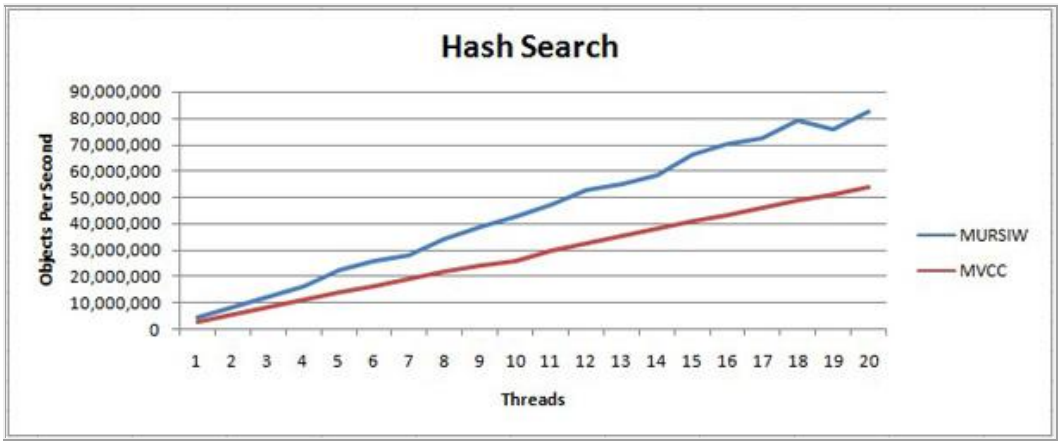

Figure 11: This graph shows the number of objects per second that can be returned using a hash index search

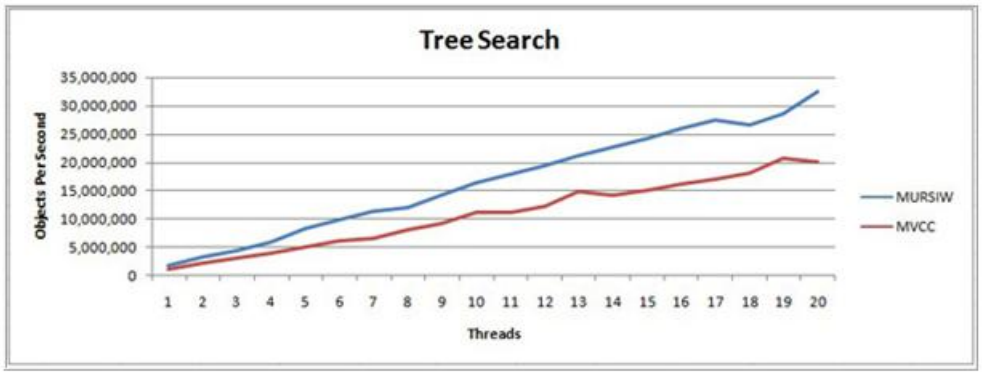

Figure 12: Performance (objects-per-second returned) using a tree index search

III. Solution & Conclusion

Recovery with concurrent transactions

Our research work has shown that when more than one transaction is being executed in parallel, the logs are interleaved. At the time of recovery it would become hard for recovery system to backtrack all logs, and then start recovering. To ease this situation most modern DBMS use the concept of 'checkpoints'.

Checkpoint

In our research work we have found that keeping and maintaining logs in real time and in real environment may fill out all the memory space available in the system. At time passes log file may be too big to be handled at all. Checkpoint is a mechanism where all the previous logs are removed from the system and stored permanently in storage disk. Checkpoint declares a point before which the DBMS was in consistent state and all the transactions were committed.

Recovery

Our research work has shown that when system with concurrent transaction crashes and recovers, it does behave in the following manner:

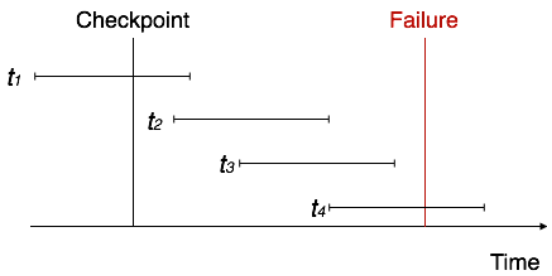

Figure 13: Recovery with concurrent transactions

The recovery system reads the logs backwards from the end to the last Checkpoint. It maintains two lists, undo-list and redo-list. If the recovery system sees a log with <T_n, Start> and <T_n, Commit> or just <T_n, Commit>, it puts the transaction in redo-list.

If the recovery system sees a log with <T_n, Start> but no commit or abort log found, it puts the transaction in undo-list. All transactions in undo-list are then undone and their logs are removed. All transaction in redo-list, their previous logs are removed and then redone again and log saved.

References

[1]. " A Secure Time-Stamp Based Concurrency Control Protocol For Distributed Databases" Journal of Computer Science 3 (7): 561-565, 2007

[2]. "Some Models of a Distributed Database Management System with Data Replication", International Conference on Computer Systems and Technologies - C*ompSysTech'07.*

[3]. "A Sophisticated introduction to distributed database concurrency control", Harvard University Cambridge, 1990.

[4]. "Database system concepts",from Silberschatz Mc-graw Hill 2001.

[5]. P. Bernstein and N. Goodman. Multiversion Concurrency Control–Theory and Algorithms. ACM TODS, 8(4):465–483, 1983.

[6]. C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In PACT, PACT '08, 2008.

[7]. R. L. Bocchino, Jr., V. S. Adve, S. V. Adve, and M. Snir. Parallel Programming Must be Deterministic by Default. HotPar, 2009.

[8]. S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-Scalable Locks are Dangerous. In Proceedings of the Linux Symposium, Ottawa, Canada, July 2012.

[9]. S. Burckhardt, A. Baldassin, and D. Leijen. Concurrent Programming with Revisions and Isolation Types. OOPSLA '10, 2010.

[10]. C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software Transactional Memory: Why Is It Only a Research Toy? Queue, 6:46–58, September 2008.

[11]. P. Cederqvist, R. Pesch, et al. Version Management with CVS. Available online with the CVS package. Signum Support AB, 1992.

[12]. J. chung, W. Baek, and C. Kozyrakis. Fast Memory Snapshot for Concurrent Programming without Synchronization. ICS, 2009.

[13]. B. Collins-Sussman, B. Fitzpatrick, and C. Pilato. Version Control with Subversion. O'Reilly Media, Incorporated, 2004.