Database Management System

Concurrency Control

Dr. Nachiyappan S

VIT-Chennai

CSE2004 Syllabus

Module-5

- Two-Phase Locking Techniques for Concurrency Control
- Concurrency Control based on timestamp
- Recovery Techniques
- Recovery based on deferred update
- Recovery techniques based on immediate update
- Shadow paging

Text Books and References

Text Books

- R. Elmasri & S. B. Navathe, Fundamentals of Database Systems, Addison Wesley, 7 th Edition, 2015
- Raghu Ramakrishnan,Database Management Systems,Mcgraw-Hill,4 th edition,2015

References

- A. Silberschatz, H. F. Korth & S. Sudershan, Database System Concepts, McGraw Hill, 6 th Edition 2010
- Thomas Connolly, Carolyn Begg, Database Systems : A Practical Approach to Design, Implementation and Management,6 th Edition,2012
- Pramod J. Sadalage and Marin Fowler, NoSQL Distilled: A brief guide to merging world of Polyglot persistence, Addison Wesley, 2012.
- Shashank Tiwari, Professional NoSql, Wiley, 2011.

VIT Vellore Institute of Technology

Introduction to Concurrency Control

- Concurrency control is the process of managing simultaneous execution of transactions (such as queries, updates, inserts, deletes and so on) in a multiprocessing database system without having them interfere with one another.
- This property of DBMS allows many transactions to access the same database at the same time without interfering with each other.
- The primary goal of concurrency is to ensure the atomicity of the execution of transactions in a multi-user database environment.
- Concurrency controls mechanisms attempt to interleave (parallel) READ and WRITE operations of multiple transactions so that the interleaved execution yields results that are identical to the results of a serial schedule execution.

Problems in Concurrency Control

When concurrent transactions are executed in an uncontrolled manner, several problems can occur.

The concurrency control has the following three main problems:

- Lost updates.
- Dirty read (or uncommitted data).
- Unrepeatable read (or inconsistent retrievals).

Lost Update Problem

A lost update problem occurs when two transactions that access the same database items have their operations in a way that makes the value of some database item incorrect.

In other words, if transactions T1 and T2 both read a record and then update it, the effects of the first update will be overwritten by the second update.

Example : Consider the figure that shows operations performed by two transactions (A and B) w.r.t Time 't'.

Transaction- A	**Time**	**Transaction- B**
——	t_0	—-
Read X	t_1	—-
—-	t_2	Read X
Update X	t_3	—-
—-	t_4	Update X
—-	t_5	—-

Lost Update Problem

Example : Consider the figure that shows operations performed by two transactions (A and B) w.r.t Time 't'.

Transaction- A	Time	Transaction- B
——	t_0	—-
Read X	t_1	—-
—-	t_2	Read X
Update X	t_3	—-
—-	t_4	Update X
—-	t_5	—-

- At t_1 , Transactions-A reads value of X.
- At t_2 , Transactions-B reads value of X.
- At t_3,Transactions-A writes value of X on the basis of the value seen at t_1.
- At t_4,Transactions-B writes value of X on the basis of the value seen at t_2.

So,update of Transactions-A is lost at time t4,because Transactions-B overwrites it without looking at its current value. (This reffered as Update Lost)

Dirty Read Problem

A dirty read problem occurs when one transaction updates a database item and then the transaction fails for some reason.The updated database item is accessed by another transaction before it is changed back to the original value.

In other words, a transaction T_1 updates a record, which is read by the transaction T_2. Then T_1 aborts and T_2 now has values which have never formed part of the stable database.

Example: Consider the below figure :

Transaction A	**Time**	**Transaction B**
—-	t_0	—-
—-	t_1	Update X
Read X	t_2	—-
—-	t_3	Rollback
—-	t_4	—-

Dirty Read Problem

Example: Consider the below figure :

Transaction A	Time	Transaction B
—-	t_0	—-
—-	t_1	Update X
Read X	t_2	—-
—-	t_3	Rollback
—-	t_4	—-

- At t_1 , Transactions-B writes value of X.
- At t_2 , Transactions-A reads value of X.
- At t_3 , Transactions-B rollbacks.So,it changes the value of X back to that of prior to t_1.

So,Transaction-A now has value which has never become part of the stable database. (Referred as Dirty Read Problem, as one transaction reads a dirty value which has not been committed.)

Inconsistent Retrievals Problem :

Unrepeatable read (or inconsistent retrievals) occurs when a transaction calculates some summary (aggregate) function over a set of data while other transactions are updating the data.

The problem is that the transaction might read some data before they are changed and other data after they are changed, thereby yielding inconsistent results.

In an unrepeatable read, the transaction T_1 reads a record and then does some other processing during which the transaction T_2 updates the record. Now, if T_1 rereads the record, the new value will be inconsistent with the previous value.

Consider the situation given in figure that shows two transactions operating on three accounts :

Account-1	Account-2	Account-3
Balance = 200	Balance = 250	Balance = 150

Inconsistent Retrievals Problem

Example:

Transaction- A	Time	Transaction- B
——	t_0	—-
Read Balance of Acc-1 Sum $\leftarrow$ 200 Read Balance of Acc-2	t_1	—-
Sum $\leftarrow$ Sum + 250 = 450	t_2	—-
—-	t_3	Read Balance of Acc-3
—-	t_4	Update Balance of Acc-3 150 $\rightarrow$ (150 - 50) $\rightarrow$ 100
—-	t_5	Read Balance of Acc-1
—-	t_6	Update Balance of Acc-1 200 $\rightarrow$ (200 + 50) $\rightarrow$ 250
Read Balance of Acc-3	t_7	COMMIT
Sum $\leftarrow$ Sum + 250 = 550	t_8	—-

- Transaction-A is summing all balances;while, Transaction-B is transferring an amount 50 from Account-3 to Account-1.
- Here,the result produced by Transaction-A is 550,which is incorrect. if this result is written in database, database will be in inconsistent state, as actual sum is 600.

  (Which is inconsistent)

Concurrency Control Protocols

Concurrency control protocols ensure atomicity, isolation, and serializability of concurrent transactions.

The concurrency control protocol can be divided into three categories:

- Lock based protocol
- Time-stamp protocol
- Validation based protocol

1. Lock Based Protocols

Any transaction cannot read or write data until it acquires an appropriate lock on it. There are two types of lock:

- Shared lock:
  1. It is also known as a Read-only lock, the data item can only read by the transaction.
  2. It can be shared between the transactions because when the transaction holds a lock, then it can't update the data on the data item.

- Exclusive lock:
  1. In the exclusive lock, the data item can be both reads as well as written by the transaction.
  2. This lock is exclusive, and in this lock, multiple transactions do not modify the same data simultaneously.

Lock Based Protocols

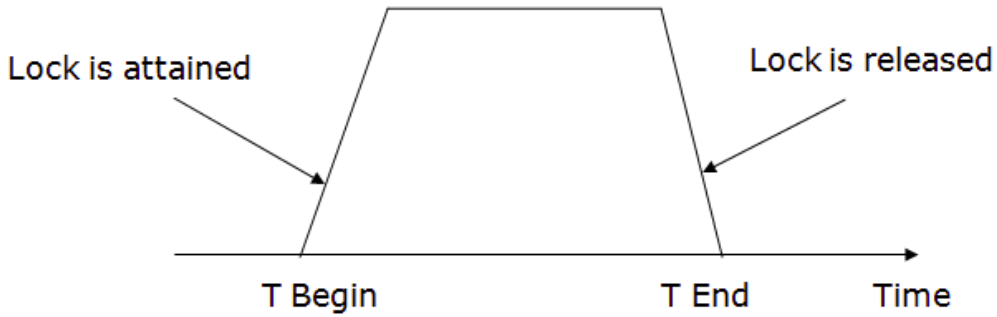
There are four types of lock protocols available:

- Simplistic lock protocol :
  1. It is the simplest way of locking the data while transaction
  2. It allow all the transactions to get the lock on the data before insert or delete or update on it.
  3. It will unlock the data item after completing the transaction.

- Pre-claiming Lock Protocol:
  1. It evaluate the transaction to list all the data items on which they need locks.
  2. Before initiating an execution of the transaction, it requests DBMS for all the lock on all those data items.
  3. If all the locks are granted then this protocol allows the transaction to begin. When the transaction is completed then it releases all the lock.
  4. If all the locks are not granted then this protocol allows the transaction to rolls back and waits until all the locks are granted.

Lock Based Protocols

- Two-phase locking (2PL):
  1. It divides the execution phase of the transaction into three parts.
  2. In the first part, when the execution of the transaction starts, it seeks permission for the lock it requires.
  3. In the second part, the transaction acquires all the locks. The third phase is started as soon as the transaction releases its first lock.
  4. In the third phase, the transaction cannot demand any new locks. It only releases the acquired locks.



Lock Based Protocols

There are two phases of 2PL:

- Growing phase: A new lock on the data item may be acquired by the transaction, but none can be released.
- Shrinking phase: Existing lock held by the transaction may be released, but no new locks can be acquired.

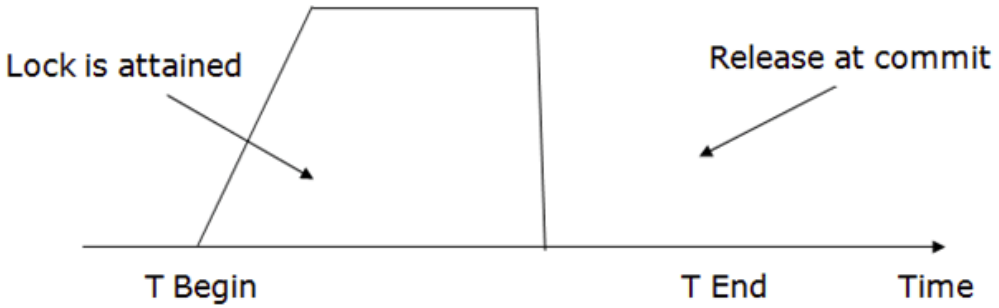
Table: Lock Conversion

	T_1	T_2
1	Lock-S(A)	
2		Lock-S(A)
3	Lock-X(B)	
4	—-	—-
5	UnLock(A)	
6		Lock-X(C)
7	UnLock(B)	
8		UnLock(A)
9		UnLock(C)
10	—	—

- Transaction T_1:
  - Growing phase: from step 1-3
  - Shrinking phase: from step 5-7
  - Lock point: at 3
- Transaction T_2:
  - Growing phase: from step 2-6
  - Shrinking phase: from step 8-9
  - Lock point: at 6

Lock Based Protocols

- Strick Two-Phase Locking (Strick-2PL)
  - The first phase of Strict-2PL is similar to 2PL. In the first phase, after acquiring all the locks, the transaction continues to execute normally.
  - The only difference between 2PL and strict 2PL is that Strict-2PL does not release a lock after using it.
  - Strict-2PL waits until the whole transaction to commit, and then it releases all the locks at a time.
  - Strict-2PL protocol does not have shrinking phase of lock release.



Note : It does not have cascading abort as 2PL does.

2. Time-Stamp Protocol

- It is used to order the transactions based on their Timestamps. The order of transaction is nothing but the ascending order of the transaction creation.
- The priority of the older transaction is higher that's why it executes first. To determine the timestamp of the transaction, this protocol uses system time or logical counter.
- The Lock-Based protocol is used to manage the order between conflicting pairs among transactions at the execution time. But Timestamp based protocols start working as soon as a transaction is created.
- Let's assume there are two transactions T_1 and T_2. Suppose the transaction T_1 has entered the system at 007 times and transaction T_2 has entered the system at 009 times. T_1 has the higher priority, so it executes first as it is entered the system first.
- The timestamp ordering protocol also maintains the timestamp of last 'read' and 'write' operation on a data.

Time-Stamp Protocol

Basic Time Stamp Protocol :

1. Check the following condition whenever a transaction T_i issues a Read(X) operation:
   - If W_TS(X) >TS(T_i) then the operation is rejected.
   - f W_TS(X) <= TS(T_i) then the operation is executed.
   - Timestamps of all the data items are updated.
2. Check the following condition whenever a transaction T_i issues a Write(X) operation:
   - If TS(T_i) < R_TS(X) then the operation is rejected.
   - If TS(T_i) < W_TS(X) then the operation is rejected and Ti is rolled back otherwise the operation is executed.

Where :

TS(TI) denotes the timestamp of the transaction T_i .
R_TS(X) denotes the Read time-stamp of data-item X.
W_TS(X) denotes the Write time-stamp of data-item X.

Mertis and DeMerits

Advantages and Disadvantages of TO protocol:

- It ensures serializability since the precedence graph is as follows:

Figure: Precedence Graph for TS Ordering

- TS protocol ensures freedom from deadlock that means no transaction ever waits.
- But the schedule may not be recoverable and may not even be cascade- free.

3. Validation Based Protocol

Validation phase is also known as optimistic concurrency control technique. In the validation based protocol, the transaction is executed in the following three phases:

1. **Read phase**: In this phase, the transaction T is read and executed. It is used to read the value of various data items and stores them in temporary local variables. It can perform all the write operations on temporary variables without an update to the actual database.

2. **Validation phase**: In this phase, the temporary variable value will be validated against the actual data to see if it violates the serializability.

3. **Write phase**: If the validation of the transaction is validated, then the temporary results are written to the database or system otherwise the transaction is rolled back.

Here each phase has the following different timestamps:

1. **Start(T_i)**: It contains the time when T_i started its execution.
2. **Validation (T_i)**: It contains the time when T_i finishes its read phase and starts its validation phase.
3. **Finish(T_i)**: It contains the time when T_i finishes its write phase.
    - This protocol is used to determine the time stamp for the transaction for serialization using the time stamp of the validation phase, as it is the actual phase which determines if the transaction will commit or rollback.
    - Hence TS(T) = validation(T).
    - The serializability is determined during the validation process. It can't be decided in advance.
    - While executing the transaction, it ensures a greater degree of concurrency and also less number of conflicts.
    - Thus it contains transactions which have less number of rollbacks.

Thanks

Database Management System

Database Recovery

Dr. Nachiyappan S

VIT-Chennai

Database Recovery in DBMS

Recovery Means... (Crash Recovery)

- A database is a very huge system with lots of data and transaction.
- The transaction in the database is executed at each seconds of time and is very critical to the database.
- If there is any failure or crash while executing the transaction, then it expected that no data is lost.
- It is necessary to revert the changes of transaction to previously committed point (Check / Restore point).
- There are various techniques to recover the data depending on the type of failure or crash.

Classification of Failure

<Employee>

EmployeeID	EmpName	DepartmentID
E09	Emily	D001
E04	Tom	D002
E11	Emma	D003

<Department>

DepartmentID	DeptName	DeptZone
D001	Finance	North
D002	Operations	East
D003	Marketing	West

Database Recovery

Recovery and Atomicity: When a DBMS recovers from a crash, it should maintain the following :

- It should check the states of all the transactions, which were being executed.
- A transaction may be in the middle of some operation; the DBMS must ensure the atomicity of the transaction in this case.
- It should check whether the transaction can be completed now or it needs to be rolled back.
- No transactions would be allowed to leave the DBMS in an inconsistent state.
- For Eg : Consider transaction T_i that transfers \$50 from account A to account B; goal is either to perform all database modifications made by T_i or none at all.

Several output operations may be required for T_i (to output A and B).

A failure may occur after one of these modifications have been made but before all of them are made

Recovery Algorithms in DBMS

Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability despite of failures

- Actions taken during normal transaction processing to ensure enough information exists to recover from failures
- Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability

There are two types of techniques, which can help a DBMS inrecovering as well as maintaining the atomicity of a transaction

- Maintaining the logs of each transaction, and writing them onto some stable storage before actually modifying the database(Log-Based Recovery).
- Maintaining shadow paging, where the changes are done on a volatile memory, and later, the actual database is updated.

Storage Structure

There are 3 types of Storage Structures :

- **Volatile (RAM/Cache)** : Doesn't Survive System crashes
- **Non-Volatile (Disks/ROM)** : Survives System Crashes
- **Stable** : Mythical form of storage, survives all failures
  - It is a classification of computer data storage technology that guarantees atomicity for any given write operation and allows software to be written that is robust against some hardware and power failure

Recovery Algorithms

Log Based Recovery : A log is kept on stable storage. i.e., a sequence of log records, which maintains a record of update activities on database.

- When Transaction T i starts, it registers itself by writing $<T_i$, Start$>$.
- Before T_i executes write(x), a log record $<T_i$, X, V_1, $V_2>$ is written, Where V_1 is the value of X before the write, and V_2 is the value to be written to X.
- When T_i finishes it last statement, the log record $< T_i$, Commit$>$ is written.
- We assume for now that log records are written directly to stable storage (that is, they are not buffered)

Two approaches

1. Deferred database modification
2. Immediate database modification.

Deferred database Modification:

- It records all modifications to the log, but defers all the writes to after partial commit.
- Assume that transactions execute serially and starts by writing $<T_i$, Start$>$ record to log.
- A write(X) operation results in a log record $<T_i$, X, V$>$ being written, where V is the new value for X
- The write is not performed on X at this time, but is deferred.
- When T_i partially commits, $<T_i$, Commit$>$ is written to the log
- Finally, the log records are read and used to actually execute the previously deferred writes.
- During recovery after a crash, a transaction needs to be redone iff both $<T_i$, Start$>$ and $<T_i$, Commit$>$ are there in the log.
- Redoing a transaction T_i (redo T_i) sets the value of all data items updated by the transaction to the new values.
- Crashes can occur while — The transaction is executing the original updates, or — While recovery action is being taken

Eg : Consider the Transaction T_0 and T_1 (T_0 executes before T_1)

T_0
read (A)
A:=A-50
write(A)
read(B)
B:=B+50
write(B)

T_1
read (C)
C:=C-100
write(C)

Log Files

$< T_0$, Start $>$	$< T_0$, Start $>$	$< T_0$, Start $>$
$< T_0$, A, 950 $>$	$< T_0$, A, 950 $>$	$< T_0$, A, 950 $>$
$< T_0$, B, 2040 $>$	$< T_0$, B, 2040 $>$	$< T_0$, B, 2040 $>$
	$< T_0$, commit $>$	$< T_0$, commit $>$
	$< T_1$, start $>$	$< T_1$, start $>$
	$< T_1$, C, 600 $>$	$< T_1$, C, 600 $>$
		$< T_1$, commit $>$
(a)	(b)	(c)

If log on stable storage at time of crash is as in case:

- (a) No redo actions need to be taken
- (b) redo(T_0) must be performed since $<T_0$,commit$>$ is present
- (c) redo(T_0) must be performed followed by redo(T_1) since $<T_0$, commit$>$ and $<T_1$, commit$>$ are present.

Immediate database Modification:

- It allows database updates of an uncommitted transaction to be made as the writes are issued.
  — since undoing may be needed, update logs must have both old value and new value.
- update log record must be written before database item is written.
  — We assume that the log record is output directly to stable storage
  — Can be extended to postpone log record output, so long as prior to execution of an output(B) operation for a data block B, all log records corresponding to items B must be flushed to stable storage
- Output of updated blocks can take place at any time before or after transaction commit
- Order in which blocks are output can be different from the order in which they are written

Immediate database Modification:

Log	Write	Output
<T_0 **start**>		
<T_0, A, 1000, 950>		
T_o, B, 2000, 2050		
	$A = 950$	
	$B = 2050$	
<T_0 **commit**>		
<T_1 **start**>		
<T_1, C, 700, 600>		
	$C = 600$	
		B_B, B_C
<T_1 **commit**>		
		B_A

Note : B_X denotes Block Containing X

Immediate database Modification:

- Recovery procedure has two operations instead of one:
  - undo(T_i) restores the value of all data items updated by T_i to their old values, going backwards from the last log record for T_i
  - redo(T_i) sets the value of all data items updated by T_i to the new values, going forward from the first log record for T_i
- Both operations must be idempotent
  - i.e., even if the operation is executed multiple times the effect is the same as if it is executed once – Needed since operations may get re-executed during recovery
- When recovering after failure:
  - Transaction T_i needs to be undone if the log contains the record $<T_i$, start$>$, but does not contain the record $<T_i$, commit$>$.
  - Transaction T_i needs to be redone if the log contains both the record $<$Ti, start$>$ and the record $<T_i$, commit$>$.
- Undo operations are performed first, then redo operations.

Immediate database Modification:

(a)	(b)	(c)
$<T_0$ start>	$<T_0$ start>	$<T_0$ start>
$<T_0, A, 1000, 950>$	$<T_0, A, 1000, 950>$	$<T_0, A, 1000, 950>$
$<T_0, B, 2000, 2050>$	$<T_0, B, 2000, 2050>$	$<T_0, B, 2000, 2050>$
	$<T_0$ commit>	$<T_0$ commit>
	$<T_1$ start>	$<T_1$ start>
	$<T_1, C, 700, 600>$	$<T_1, C, 700, 600>$
		$<T_1$ commit>

Recovery Actions :

- **undo(T_0)** : B is restored to 2000 and A to 1000.
- **undo(T_1)** and **redo(T_0)** : C is restored to 700 and the A & B to 950, 2050 Respectively.
- **redo(T_0)** and **redo(T_1)** : A & B are set to 950, 2050 Respectively. Then C is set to 600

Shadow Paging Recovery

It is an alternative to log-based recovery techniques, which has both advantages and disadvantages.

It may require fewer disk accesses, but it is hard to extend paging to allow multiple concurrent transactions.

The paging is very similar to paging schemes used by the operating system for memory management.

Recovery Algorithms

Shadow Paging Recovery

- The idea is to maintain two page tables during the life of a transaction: the **current page** table and the **shadow page** table.
- When the transaction starts, both tables are **identical**.
- The **shadow page** is never changed during the life of the transaction.
- The **current page** is updated with each write operation.
- Each table entry points to a page on the disk.
- When the transaction is committed, the **shadow page** entry becomes a copy of the **current page** table entry and the disk block with the old data is released.
- If the shadow is stored in nonvolatile memory and a system crash occurs, then the **shadow page** table is copied to the **current page** table.
- This guarantees that the shadow page table will point to the database pages corresponding to the state of the database prior to any transaction that was active at the time of the crash, making aborts automatic.

There are drawbacks to the shadow-page technique:

- **Commit** overhead: The commit of a single transaction using shadow paging requires multiple blocks to be output.(i.e., the current page table, the actual data and the disk address of the current page table). Log-based schemes need to output only the log records.
- **Data fragmentation**: Shadow paging causes database pages to change locations (therefore, no longer contiguous).
- **Garbage collection**: Each time that a transaction commits, the database pages containing the old version of data changed by the transactions must become inaccessible. Such pages are considered to be garbage since they are not part of the free space and do not contain any usable information. Periodically, need to find all such pages and add them to the list of free pages. This process is called garbage collection and imposes additional overhead and complexity on the system.

Advantages :

- No Overhead for writing log records.
- No Undo / No Redo algorithm.
- Recovery is faster.

DisAdvantages :

- Data gets fragmented or scattered.
- After every transaction completion database pages containing old version of modified data need to be garbage collected.
- Hard to extend algorithm to allow transaction to run concurrently

Thanks