Transaction Management

Chapter 3

What is a Transaction?

- A logical unit of work on a database
  - An entire program
  - A portion of a program
  - A single command
- The entire series of steps necessary to accomplish a logical unit of work
- Successful transactions change the database from one CONSISTENT STATE to another

  (One where all data integrity constraints are satisfied)

Example of a Transaction

- Updating a Record
  - Locate the Record on Disk
  - Bring record into Buffer
  - Update Data in the Buffer
  - Writing Data Back to Disk

4 Properties of a Transaction

- Atomic – All or Nothing

  All parts of the transaction must be completed and committed or it must be aborted and rolled back

- Consistent

  Each user is responsible to ensure that their transaction (if executed by itself) would leave the database in a consistent state

4 Properties of a Transaction

- Isolation

  It indicates that action performed by a transaction will be hidden from outside the transaction until the transaction terminates.

- Durability

  If a transaction has been committed, the DBMS must ensure that its effects are permanently recorded in the database (even if the system crashes)

(ACID properties of transaction)

Transaction Management with SQL

- SQL Statements → Commit / Rollback
- When a transaction sequence is initiated it must continue through all succeeding SQL statements until:
  1. A Commit Statement is Reached
  2. A Rollback Statement is Reached
  3. The End of the Program is Reached (Commit)
  4. The Program is Abnormally Terminated (Rollback)

Example

```
BEGIN TRAN
    DECLARE @ErrorCode INT, @TranSuccessful INT
    SET @TranSuccessful = 1

    INSERT INTO tblCatalog (CatalogYear)
        VALUES('2002')
    SET @ErrorCode = @@ERROR; IF (@ErrorCode <> 0) SET @TranSuccessful = 0 –
    False

    INSERT INTO tblCatalog (CatalogYear)
        VALUES('2003')
    SET @ErrorCode = @@ERROR; IF (@ErrorCode <> 0) SET @TranSuccessful = 0 –
    False

    IF @TranSuccessful = 0
      BEGIN
        ROLLBACK TRAN
        RAISERROR ('Rolledback transaction: Insert Catalog Year.', 16,1)
      END
    ELSE
      BEGIN
        COMMIT TRAN
        PRINT 'Successfully inserted catalog years...'
      END
GO
```

Transaction Log

- Keeps track of all transactions that update the database
  - Record for the beginning of the transaction
  - Type of operation (insert / update / delete)
  - Names of objects/tables affected by the transaction
  - Before and After Values for Updated Fields
  - Pointers to Previous and Next Transaction Log Entries for the same transaction
  - The Ending of the Transaction (Commit)
- Used for recovery in case of a Rollback

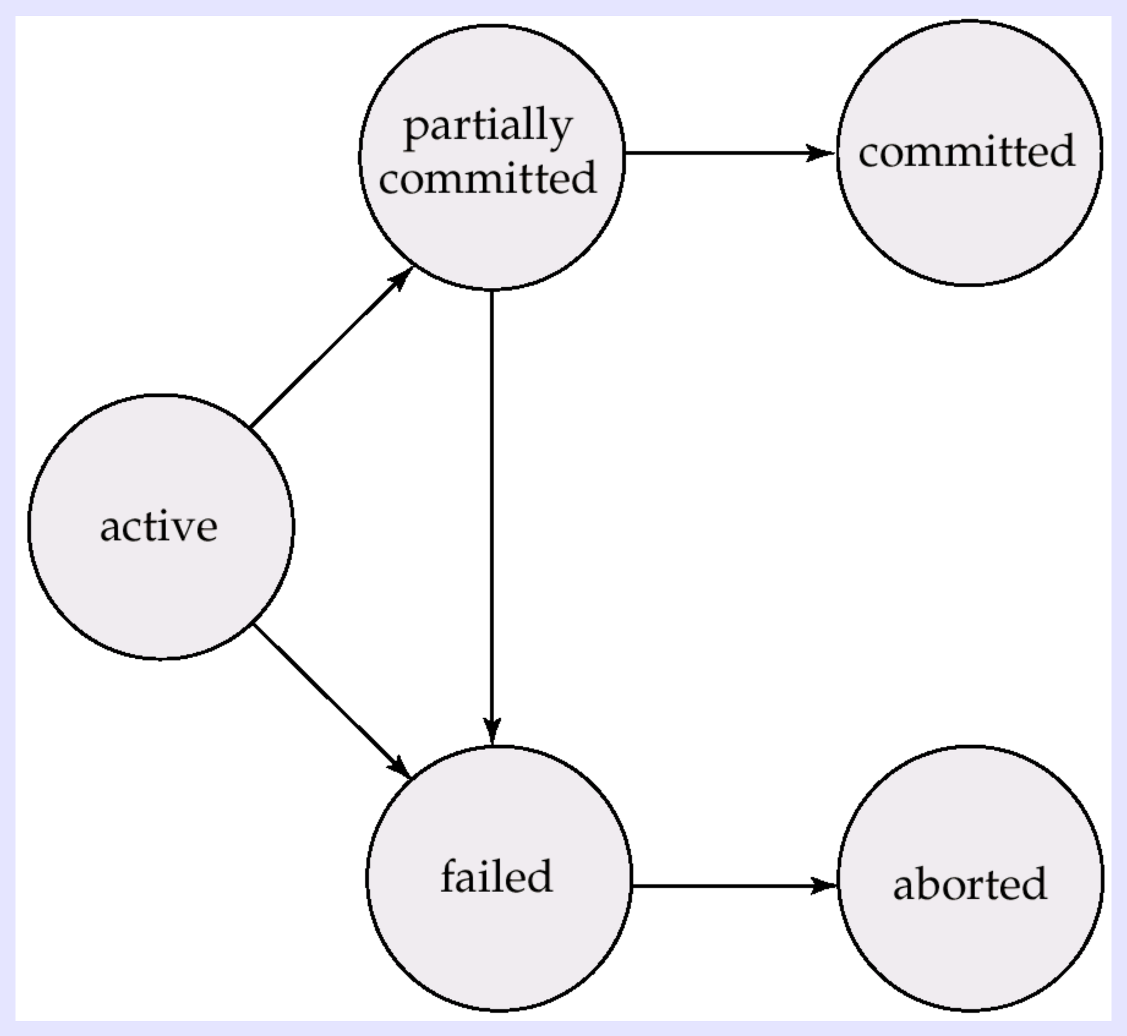
Transaction States

- Because failures occurs, transaction are broken up into states to handle various situation.

1. Active : The initial state; the transaction stays in this state until while it is still executing.
2. Partially Committed : After the final statement has been executed
3. Failed : after the discovery that normal execution can no longer proceed
4. Committed : after successful completion.

5. Aborted : after the transaction has been rolled back the the database has been restored to its state prior to the start of the transaction.

State diagram of a transaction



Schedules

- *Schedules* – sequences that indicate the chronological order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.

Example Schedules

- Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B. The following is a serial schedule (Schedule 1 in the text), in which T_1 is followed by T_2.

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Cont.

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1.

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

Cont.

- The following concurrent schedule does not preserve the value of the the sum $A + B$.

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

Serializability

- Basic Assumption – Each transaction preserves database consistency.
- Thus serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:
  1. conflict serializability
  2. view serializability

Conflict Serializability

- Instructions l_i and l_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both l_i and l_j, and at least one of these instructions wrote Q.

1. l_i = **read**(Q), l_j = **read**(Q). l_i and l_j don't conflict.
2. l_i = **read**(Q), l_j = **write**(Q). They conflict.
3. l_i = **write**(Q), l_j = **read**(Q). They conflict
4. l_i = **write**(Q), l_j = **write**(Q). They conflict

Conflict Serializability (Cont.)

- If a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, we say that S and S' are **conflict equivalent**.
- We say that a schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule
- Example of a schedule that is not conflict serializable:

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

We are unable to swap instructions in the above schedule to obtain either the serial schedule $< T_3, T_4 >$, or the serial schedule $< T_4, T_3 >$.

Conflict Serializability (Cont.)

- Schedule 3 below can be transformed into Schedule 1, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore Schedule 3 is conflict serializable.

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

View Serializability

- Let S and S' be two schedules with the same set of transactions. S and S' are **view equivalent** if the following three conditions are met:
  1. For each data item Q, if transaction T_i reads the initial value of Q in schedule S, then transaction T_i must, in schedule S', also read the initial value of Q.
  2. For each data item Q if transaction T_i executes **read**(Q) in schedule S, and that value was produced by transaction T_j (if any), then transaction T_i must in schedule S' also read the value of Q that was produced by transaction T_j .
  3. For each data item Q, the transaction (if any) that performs the final **write**(Q) operation in schedule S must perform the final **write**(Q) operation in schedule S'.

As can be seen, view equivalence is also based purely on **reads** and **writes** alone.

View Serializability (Cont.)

- A schedule *S* is **view serializable** it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Schedule 9 (from text) — a schedule which is view-serializable but *not* conflict serializable.

  Every view serializable schedule that is not conflict serializable has **blind writes.**

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

Recoverable Schedules

Need to address the effect of transaction failures on concurrently running transactions.

- **Recoverable schedule** — if a transaction T_j reads a data items previously written by a transaction T_i, the commit operation of T_i appears before the commit operation of T_j.
- The following schedule is not recoverable if T_9 commits immediately after the read.
- If T_8 should abort, T_9 would have read (and possibly shown to the user) an inconsistent database state. Hence database must ensure that schedules are recoverable

T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

Cascading Schedule

- Every cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless
- **Cascadeless schedules** — cascading rollbacks cannot occur; for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.

Cascadeless Schedules (Cont.)

- **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)
- If T_{10} fails, T_{11} and T_{12} must also be rolled back.
- Can lead to the undoing of a significant amount of work

Concurrency Control

- Coordination of simultaneous transaction execution in a multiprocessing database system
- Ensure transaction serializability in a multi-user database
- Lack of Concurrency Control can create data integrity and consistency problems:
  - Lost Updates
  - Uncommitted Data
  - Inconsistent Retrievals

Lost Updates

Time	Jack's Trans	Jill's Trans	Balance
T1	Begin		
T2	Read Balance	Begin	1000
T3		Read Balance	1000
T4	Bal = Bal – 50 (950)		1000
T5	Write Bal (950)	Bal = Bal + 100 (1100)	950
T6	Commit		950
T7		Write Bal (1100)	1100
T8		Commit	1100

Uncommitted Data

Time	Deposit	Interest	Bal
T1	Begin Transaction		1000
T2	Read Bal (1000)		1000
T3	Bal = Bal + 1000 (2000)		1000
T4	Write Bal (2000)	Begin Transaction	2000
T5		Read Bal (2000)	2000
T6		Bal = Bal*1.05 (2100)	2000
T7	Rollback		1000
T8		Write Bal (2100)	2100
T9		Commit	2100

Inconsistent Retrievals

Time	SumBal	Transfer	Bal A	Bal B	Bal C	Sum
T1	Begin Trans		5000	5000	5000	
T2	Sum = 0	Begin Trans	5000	5000	5000	
T3	Read BalA (5000)		5000	5000	5000	
T4	Sum = Sum + BalA (5000)	Read BalA (5000)	5000	5000	5000	
T5	Read BalB (5000)	BalA = BalA -1000 (4000)	5000	5000	5000	
T6	Sum = Sum+BalB (10000)	Write BalA (4000)	4000	5000	5000	
T7		Read BalC	4000	5000	5000	
T8		BalC =BalC + 1000 (6000)	4000	5000	5000	
T9		Write BalC (6000)	4000	5000	6000	
T10	Read BalC	Commit	4000	5000	6000	
T11	Sum=Sum + BalC (16000)		4000	5000	6000	
T12	Write Sum (16000)		4000	5000	6000	16000
T13	Commit		4000	5000	6000	16000

Serial Execution of Transactions

- Serial Execution of transaction means that the transactions are performed one after another.
- No interaction between transactions - No Concurrency Control Problems
- Serial Execution will never leave the database in an inconsistent state → Every Serial Execution is considered correct (Even if a different order would cause different results)

Serializability

- If 2 Transactions are only reading data items – They do not conflict → Order is unimportant
- If 2 Transactions operate (Read/Write) on Separate Data Items
  - They do not conflict → Order is unimportant
- If 1 Transaction Writes to a Data Item and Another Reads or Writes to the Same Data Item → The Order of Execution IS Important

The Scheduler

- Special DBMS Program to establish the order of operations in which concurrent transactions are executes
- Interleaves the execution of database operations to ensure:

  Serializability

  Isolation of Transactions

The Scheduler

- Bases its actions on Concurrency Control Algorithms (Locking / Time Stamping)
- Ensures the CPU is used efficiently (Scheduling Methods)
- Facilitates Data Isolation → Ensure that 2 transactions do not update the same data at the same time

Concurrency Control Algorithms

- Locking

  A Transaction “locks” a database object to prevent another object from modifying the object

- Time-Stamping

  Assign a global unique time stamp to each transaction

- Optimistic

  Assumption that most database operations do not conflict

Locking

- Lock guarantees exclusive use of data item to current transaction
- Prevents reading Inconsistent Data
- Lock Manager is responsible for assigning and policing the locks used by the transaction

Locking Granularity

Indicates the level of lock use

- Database Level – Entire Database is Locked
- Table Level – Entire Table is Locked
- Page Level – Locks an Entire Diskpage (Most Frequently Used)
- Row Level – Locks Single Row of Table
- Field Level – Locks a Single Attribute of a Single Row (Rarely Done)

Types of Locks: Binary

- Binary Locks – Lock with 2 States
  - Locked – No other transaction can use that object
  - Unlocked – Any transaction can lock and use object

All Transactions require a Lock and Unlock Operation for Each Object Accessed (Handled by DBMS)

- Eliminates Lost Updates
- Too Restrictive to Yield Optimal Concurrency Conditions

Types of Locks: Shared / Exclusive Locks

- Indicates the Nature of the Lock
- Shared Lock – Concurrent Transactions are granted READ access on the basis of a common lock
- Exclusive Lock – Access is reserved for the transaction that locked the object
- 3 States: Unlocked, Shared (Read), Exclusive (Write)
- More Efficient Data Access Solution
- More Overhead for Lock Manager
  - Type of lock needed must be known
  - 3 Operations:
    - Read_Lock – Check to see the type of lock
    - Write_Lock – Issue a Lock
    - Unlock – Release a Lock
  - Allow Upgrading / Downgrading of Locks

Problems with Locking

- Transaction Schedule May Not be Serializable
  - Can be solved with 2-Phase Locking
- May Cause Deadlocks
  - A deadlock is caused when 2 transactions wait for each other to unlock data

Two Phase Locking

- Defines how transactions Acquire and Relinquish Locks

1. Growing Phase – The transaction acquires all locks (doesn't unlock any data)
2. Shrinking Phase – The transaction releases locks (doesn't lock any additional data)

- Transactions acquire all locks it needs until it reaches locked point
- When locked, data is modified and locks are released

Deadlocks

- Occur when 2 transactions exist in the following mode:
  T1 = access data item X and Y
  T2 = Access data items Y and X

If T1 does not unlock Y, T2 cannot begin
If T2 does not unlock X, T1 cannot continue

T1 & T2 wait indefinitely for each other to unlock data

- Deadlocks are only possible if a transactions wants an Exclusive Lock (No Deadlocks on Shared Locks)

Controlling Deadlocks

- Prevention – A transaction requesting a new lock is aborted if there is the possibility of a deadlock – Transaction is rolled back, Locks are released, Transaction is rescheduled
- Detection – Periodically test the database for deadlocks. If a deadlock is found, abort / rollback one of the transactions
- Avoidance – Requires a transaction to obtain all locks needed before it can execute – requires locks to be obtained in succession

Time Stamping

- Creates a specific order in which the transactions are processed by the DBMS
- 2 Main Properties
  1. Uniqueness – Assumes that no equal time stamp value can exist (ensures serializability of the transactions)
  2. Monotonicity – Ensures that time stamp values always increases
- All operations within the same transaction have the same time stamp
- If Transactions conflict, one is rolled back and rescheduled
- Each value in Database requires 2 Additional Fields: Last Time Read / Last Time Updated
- Increases Memory Need and Processing Overhead

Time Stamping Schemes

- Wait / Die Scheme
  - The older transaction will wait
  - The younger transaction will be rolled back
- Wound / Wait Scheme
  - The older transaction will preempt (wound) the younger transaction and roll it back
  - The younger transaction waits for the older transaction to release the locks
- Without time-out values, Deadlocks may be created

Optimistic Method

- Most database operations do not conflict
- No locking or time stamping
- Transactions execute until commit
  - Read Phase – Read database, execute computations, make local updates (temporary update file)
  - Validate Phase – Transaction is validated to ensure changes will not effect integrity of database
    - If Validated → Go to Write Phase
    - If Not Validated → Restart Transaction and discard initial changes
  - Write Phase – Commit Changes to database
- Good for Read / Query Databases (Few Updates)

Database Recovery

- Restore a database from a given state to a previous consistent state
- Atomic Transaction Property (All or None)
- Backup Levels:
  - Full Backup
  - Differential Backup
  - Transaction Log Backup
- Database / System Failures:
  - Software (O.S., DBMS, Application Programs, Viruses)
  - Hardware (Memory Chips, Disk Crashes, Bad Sectors)
  - Programming Exemption (Application Program rollbacks)
  - Transaction (Aborting transactions due to deadlock detection)
  - External (Fire, Flood, etc)

Transaction Recovery

- Recover Database by using data in the Transaction Log
- Write-Ahead-Log – Transaction logs need to be written before any database data is updated
- Redundant Transaction Logs – Several copies of log on different devices
- Database Buffers – Buffers are used to increase processing time on updates instead of accessing data on disk
- Database Checkpoints – Process of writing all updated buffers to disk → While this is taking place, all other requests are not executes
  - Scheduled several times per hour
  - Checkpoints are registered in the transaction log