Databases and Information Systems
CS303

Transaction Management
15-09-2023

Transaction

- Unit of program execution that accesses and possibly updates various data items.
- Indivisible set of instructions
  - Typically uses begin transaction and end transaction
- If a transaction fails, all its changes to the database should be undone
  - Failure could be of the OS, hardware, logical error etc.
- Should appear as a single operation
  - No effect of other concurrent operations
- Once the transaction succeeds, all its updates to the database should persist
  - Even the system fails
- A successful execution of a transaction must take a database in a consistent state to another consistent state
  - Responsibility of the programmer

ACID : Properties of Transaction

- Atomicity : Either all operations of the transaction are reflected properly in
- the database, or none are

- Consistency : Execution of a transaction in isolation preserves the consistency of the database.

- Isolation : Each transaction is unaware of other transactions executing concurrently in the system.
  - In presence of concurrent transactions, the system guarantees that for every pair of transactions T_i and T_j , it appears to T_i that either T_j finished execution before T_i started or T_j started execution after T_i finished.

- Durability : After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Simple Transaction Model

- To understand Transaction, SQL is too complex
- Let us fix a simpler language that helps us focus on when data is moved from disk to main memory and vice-versa
- Main operations:
  - read(X) : transfers the data item X from the database to a variable, also called X
    - Variable is in a buffer in main memory belonging to the transaction that executed the read operation
  - write(X) : transfers the value in the variable X in the main-memory buffer of the transaction that executed the write to the data item X in the database

Transaction Example

- Main operations:
  - read(X) : transfers the data item X from the database to a variable, also called X
    - Variable is in a buffer in main memory belonging to the transaction that executed the read operation
  - write(X) : transfers the value in the variable X in the main-memory buffer of the transaction that executed the write to the data item X in the database

- Exam transaction of transferring 50 rupees From A to B

T_i: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

Transaction Example

- Exam transaction of transferring 50 rupees From A to B

- Consistency :
  - A+B is unchanged after the execution of the transaction
  - Responsibility of the programmer who codes the transaction

T_i: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

Transaction Example

- Exam transaction of transferring 50 rupees From A to B

T_i: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

- Atomicity :
  - Suppose A = 2000, B = 3000 before execution
  - T_i fails to execute completely
  - Failure happens after write(A); before write(B)
  - Then A+B value has changed (inconsistent state)

  - System has to be at inconsistent state at some point
  - Becomes consistent once the transaction executes successfully
  - If a transaction is guaranteed to never start or execute complete, Intermediate inconsistency is not visible except during the execution

Transaction Example

- Exam transaction of transferring 50 rupees From A to B

T_i: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

- Atomicity :
  - The database system keeps track (on disk) of the old values of any data on which a transaction performs a write.
  - This information is written to a file called the log.
  - If the transaction does not complete its execution, the database system restores the old values from the log to make it appear as though the transaction never executed.
  - Ensuring atomicity is the responsibility of the database system;
    - handled by a component of the database called the recovery system

Transaction Example

- Exam transaction of transferring 50 rupees From A to B

T_i: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

- Durability :
  - If transaction completes successfully, the update should persist
  - No system failure can result in a loss of data corresponding to this transfer of funds.

  - We assume that a failure of the computer system may result in loss of data in main memory, but data written to disk are never lost.
  - Loss of data on disk (discussed later)

Transaction Example

- Exam transaction of transferring 50 rupees From A to B

T_i: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

- Durability :
  - Ensured by following one of these protocols:
    - The updates carried out by the transaction have been written to disk before the transaction completes.
    - Store sufficient information about the updates of the transaction to enable the database to reconstruct the updates when the database system is restarted after the failure.
      - Useful if system crashed after update, before changes are written to disk
  - Recovery system is responsible for ensuring Durability

Transaction Example

T_i: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

- Exam transaction of transferring 50 rupees From A to B

- Isolation :
  - If several transactions execute concurrently,
    Operations may interleave and lead to unwanted results
  - If a different transaction T_j reads values of A,B when T_i is executing
    (when it is in inconsistent state)
  - Then T_j updates the database based on an inconsistent state of the database

  - One way to ensure isolation is to execute the transactions in serial
  - But leads to performance overhead
  - There are other solutions (serializability)

  - Ensuring isolation is the responsibility of the concurrency-control-system component of the database

Storage Structures

- Which storages are permanent?
    - Volatile Storage : Main memory, Cache memory
        - Fast memory access
        - Does not survive system crash
    - Non Volatile Storage : Magnetic hard disk, USB, cloud
        - Access is slower than volatile storage
        - Survives system crash
        - Susceptible to failures
    - Stable storage : By keeping copies of data on various Non volatile storages with independent failure modes
        - Probability of failure is very close to 0
        - To be updated with care : There should not be loss of information if there is a failure during the updates

Storage Structures

- RAID (Redundant Array of Independent Disks) provide battery for memory
- For a transaction to be durable, its changes need to be written to stable storage.
- For a transaction to be atomic, log records need to be written to stable storage before any changes are made to the database on disk.
- Capacity of a system to ensure durability and atomicity depends on how stable its implementation of stable storage really is.
- A single copy on disk is sufficient for most cases
    - Cases where data is highly valuable and whose transactions are highly important require multiple copies

Atomicity and Durability : Aborted transactions

- A transaction that starts but crashes in the middle is called an Aborted Transaction

- An aborted transaction should have no effect on the database
  - Changes done by an aborted transactions to the database should be undone
  - Once the changes caused by an aborted transaction have been undone, we say that the transaction has been rolled back

- Recovery system of the database tool should handle the roll backs

Atomicity and Durability : Aborted transactions

- Rollbacks are done by keeping logs
  - Each database modification made by a transaction is recorded in the log.
  - Every entry has information about:
    - Identifier of the transaction performing the modification,
    - Identifier of the data item being modified
    - both the old value (prior to modification) and the new value (after modification)
  - Only after log is entered, the database is modified

- Logs help in
  - Redoing a modification to ensure atomicity and durability when transaction succeeds
  - Possibility of undoing a modification to ensure atomicity in case of a failure during transaction execution.

Atomicity and Durability : Committed transactions

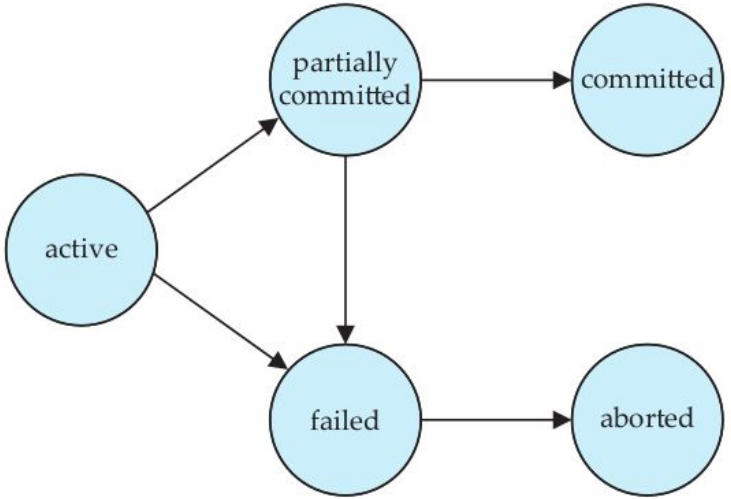
- A transaction that completes its execution successfully is said to be committed transaction.
- Committed transaction changes the state of the database from one consistent state to another
- All such changes should persist in the database (durability)
- Once a transaction has committed, we cannot undo its effects by aborting it.
  - The only way to undo the effects of a committed transaction is to execute a compensating transaction.
  - Responsibility of writing and executing a compensating transaction is left to the Programmer

Atomicity and Durability

- What does successful completion of a transaction mean?

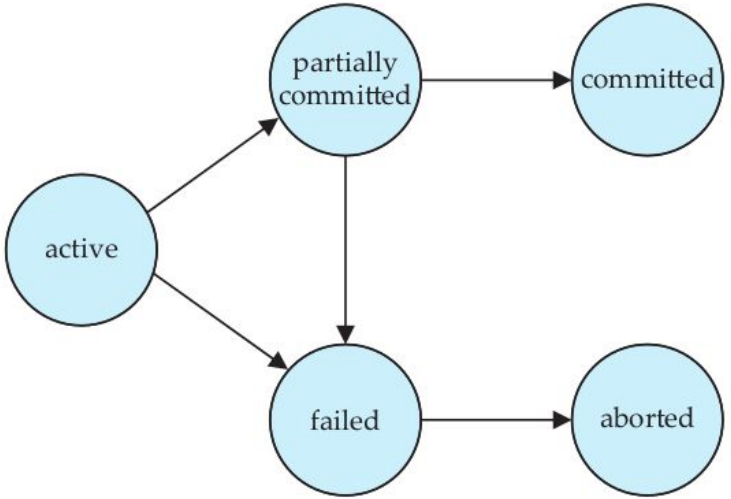
- A transaction must be in one of the following states:
  - Active or initial state; the transaction stays in this state while it is executing.
  - Partially committed : after the final statement has been executed.
  - Failed : after the discovery that normal execution can no longer proceed.
  - Aborted : after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction.
  - Committed: after successful completion.

- Transaction is terminated if it is committed or aborted



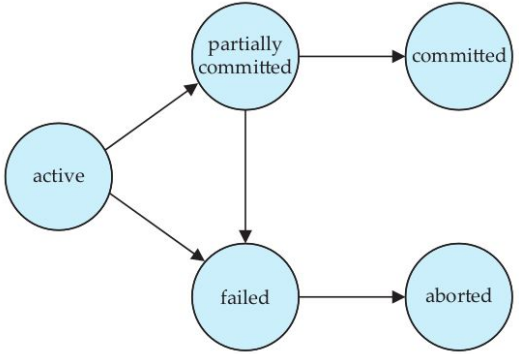
Atomicity and Durability

- A transaction starts in the active state.
- When it finishes its final statement, it enters the partially committed state.
  - Actual output may still be temporarily residing in main memory
  - The transaction has completed its execution, but it is still possible that it may have to be aborted
    - Hardware failure
- The database system then writes these updates to disk
  - When the last of this information is written out, the transaction enters the committed state.



Atomicity and Durability

- A transaction enters the failed state after the system determines that the transaction can no longer proceed with its normal execution
  - hardware or logical errors etc.
- Such a transaction must be rolled back.
  - After rolling back it enters abort state
- Two options after abort:
  - Restart the transaction
    - Only if the transaction was aborted as a result of some hardware or software error
    - Not some internal logic of the transaction.
    - A restarted transaction is considered to be a new transaction.
  - Kill the transaction



Atomicity and Durability

- Observable externals writes should always be persistent
  - Things that cannot be erased (like sent mail, printed invoice, cash dispensed at ATM etc)
- Do this only after transaction commits
- If system crashed after commit before external events occur
  - Do it when system restarts
  - Or execute compensating transaction

Isolation

- Allowing multiple transactions to update data concurrently causes several complications with consistency of the data
- Serial execution of transactions are always safe
- Advantages of interleaving execution of transactions:
  - Improved throughput and resource utilization
  - Reduced waiting time and average response time
- Concurrent transactions may result in database consistency being violated
  - Even if each individual transact preserves consistency
- The database system must control the interaction among the concurrent transactions to prevent them from violating the consistency of the database.

Isolation

- Can we identify those executions that are guaranteed to ensure the isolation property (and database consistency) ?

- Assume initially
  A = 2000
  B = 3000

T_1: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

T_2: read(A);
$temp := A * 0.1$;
$A := A - temp$;
write(A);
read(B);
$B := B + temp$;
write(B).

Isolation : Scheduling

- Assume initially
  A = 2000
  B = 3000

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

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

T_1: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

T_2: read(A);
$temp := A * 0.1$;
$A := A - temp$;
write(A);
read(B);
$B := B + temp$;
write(B).

- Both schedules preserve Consistency : A+B is constant

Isolation : Serial Schedule

- Schedule : Ordering the sequence of concurrent transactions
- Schedules preserve order on operations within each of the transaction
- Serial schedule: All operations of a single transaction appears together
- If there are n transactions how many different possible serial schedules are possible?

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

T_1: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

T_2: read(A);
$temp := A * 0.1$;
$A := A - temp$;
write(A);
read(B);
$B := B + temp$;
write(B).

Isolation : Non serial schedule

- Non-serial schedules:
  - Interleaves between transactions

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)	
commit	
	read(B)
	$B := B + temp$
	write(B)
	commit

T_1: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

T_2: read(A);
$temp := A * 0.1$;
$A := A - temp$;
write(A);
read(B);
$B := B + temp$;
write(B).

Isolation : Non serial schedule

- Non-serial schedules:
  - Interleaves between transactions

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)	
commit	
	$B := B + temp$
	write(B)
	commit

T_1: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

T_2: read(A);
$temp := A * 0.1$;
$A := A - temp$;
write(A);
read(B);
$B := B + temp$;
write(B).

- May result in inconsistent database states

Isolation : Non serial schedule

- If the scheduling is left to the OS, the interleaving could result in inconsistent database state
- Scheduling is handled by concurrency-control component of the database system

Isolation : Serializability

- An interleaving schedule is acceptable if it has the same effect of some serial schedule
- Such schedules are called serializable schedules.
- How to know which interleavings are serializable and which are not?

Isolation : Serializability

- Only read(X) and write(X) are important operations to consider

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)	
commit	
	read(B)
	$B := B + temp$
	write(B)
	commit

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

T_1: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

T_2: read(A);
$temp := A * 0.1$;
$A := A - temp$;
write(A);
read(B);
$B := B + temp$;
write(B).

Isolation : Serializability

- Consider two instructions I and J from T_i and T_j respectively
  - If I = read(X) and J = read(X)
    - Order of I and J does not matter
  - If I = read(X) and J = write(X)
    - Order of I and J matters
  - If I = write(X) and J = read(X)
    - Order of I and J matters
  - If I = write(X) and J = write(X)
    - Order of I and J matters (affects the next read(X) operation)

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

- Two instructions I and J are in conflict if
  - they belong to two different transactions
  - They access the same data item
  - At least one of them is a write operation

Isolation : Serializability

- If I and J are two consecutive instructions from different transactions and are not in conflict and in the schedule S
  - We can obtain a new schedule S' by swapping I and J
  - S and S' will be equivalent

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

- Both these schedules are Conflict Equivalent
  - Order of execution of conflicting instructions are the same
- Hence the initial schedule is Conflict serializable

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

Isolation : Serializability

- A schedule S is conflict serializable if S is conflict equivalent to some serial schedule

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

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

Not conflict serializable

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

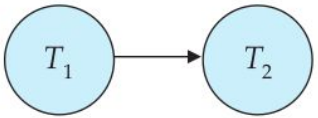
Isolation : Serializability

- Algorithm to determine conflict serializability of a schedule
  - Given a schedule S construct a precedence graph G(V,E) as follows:
    - V = all transactions of the schedule
    - E contains a directed edge from T_i to T_j if:
      - T_i executes write(X) before T_j executes read(X)
      - T_i executes read(X) before T_j executes write(X)
      - T_i executes write(X) before T_j executes write(X)

    (Edge from T_i to T_j says that T_i should be executed before T_j)

  - If precedence graph of S contains a cycle then S is NOT serializable
  - If precedence graph of S does not contain a cycle then S is serializable

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



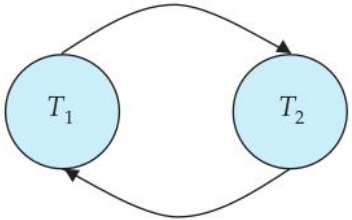
Isolation : Serializability

- Algorithm to determine conflict serializability of a schedule

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)	
commit	
	$B := B + temp$
	write(B)
	commit

  - Given a schedule S construct a precedence graph G(V,E) as follows:
    - V = all transactions of the schedule
    - E contains a directed edge from T_i to T_j if:
      - T_i executes write(X) before T_j executes read(X)
      - T_i executes read(X) before T_j executes write(X)
      - T_i executes write(X) before T_j executes write(X)

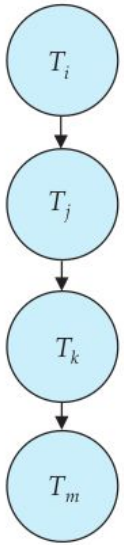
    (Edge from T_i to T_j says that T_i should be executed before T_j)

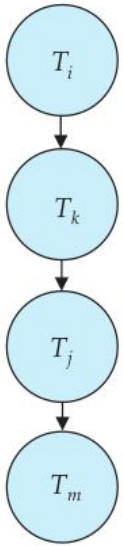


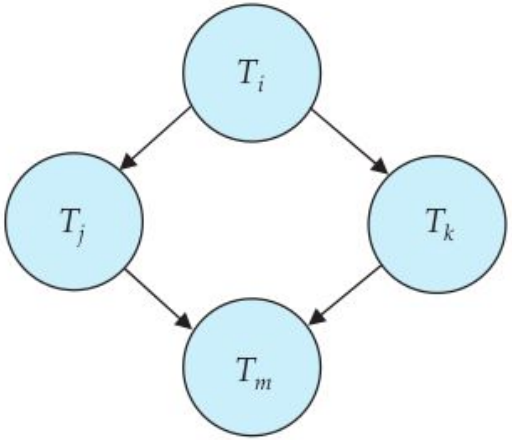
  - If precedence graph of S contains a cycle then S is NOT serializable
  - If precedence graph of S does not contain a cycle then S is serializable

Isolation : Serializability

- Serializability order can be obtained by topological sorting of directed acyclic graphs

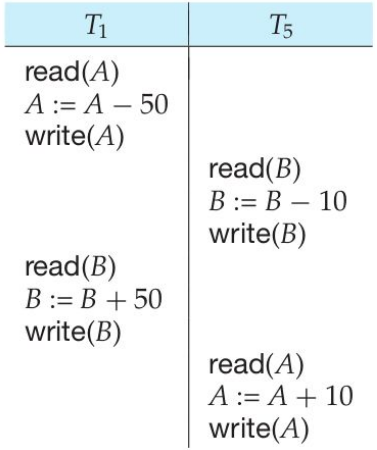






Isolation : Serializability

- Two schedules that are not conflict equivalent can still produce same result
- This schedule is not serializable
- But final result obtained will be same as that obtained by executing T_1 followed by T_5



T_1	T_5
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

- There can be less strict notion of serializability like
  - View equivalence

Isolation and Atomicity : Recoverable schedules

- What happens to schedules when transactions fail?
- If T_i fails then we need to undo the effect of T_i over the database
- Then any transaction T_j that depends on T_i's updates should also be aborted
  - So we need to restrict the allowed schedules
    - Recoverable and Cascadeless schedules

T_6	T_7
read(A)	
write(A)	
	read(A)
	commit
read(B)	

Non Recoverable schedule if T_6 aborts

- Recoverable schedule : For each pair of transactions T_i and T_j
  - if T_j reads a data item previously written by T_i then the commit operation of T_i appears before the commit operation of T_j

Isolation and Atomicity : Cascadeless schedules

- Even if a schedule is recoverable, abort might have to rollback several transactions.
  - Bad since it needs significant time and computation
  - Cascadeless schedules : 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 .

- Every cascadeless schedule is also a recoverable schedule

T_8	T_9	T_{10}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)
abort		

Schedule that needs cascading rollback

Transaction Isolation levels

- Serializability ensures that it is sufficient to maintain consistency within a transaction

- But leaves little room for interleaving
  - Use weaker notions of consistency
  - Places burden on programmers to ensure database correctness

- SQL allows explicit specification:
  - Example : Operate at the level of uncommitted read
    - Permits the transaction to read a data item even if it was written by a transaction that has not been committed
    - Used for long transactions that happen concurrently

Transaction Isolation levels

- Different transaction Isolation Levels:
  - Serializable ensures serializable execution.
  - Repeatable Read allows only committed data to be read and also between two reads of a data item by a transaction, no other transaction is allowed to update it.
  - Read Committed allows only committed data to be read, but does not require repeatable reads.
  - Read Uncommitted allows uncommitted data to be read.
    - Lowest Isolation level
- All isolation levels disallow dirty reads
  - Disallow writes to a data item that has already been written by another transaction that has not yet committed or aborted.

Transaction Isolation levels

- Most database systems have Read committed as default isolation level
  - set transaction isolation level serializable;
- Changing isolation level must be done as the first statement of a transaction.
- Automatic commit of individual statements must be turned off, if it is on by default;
- How to implement Isolation levels?
  - Discussed later

Transactions as SQL Statements

- A transaction begins implicitly when an SQL statement is executed.
- End:
  - COMMIT WORK (or just COMMIT)
  - ROLLBACK WORK (or just ROLLBACK)
- If system crashes before reaching the end, either commit / rollback happens
  - Depends on implementation
- Since every SQL statement is treated as a transaction, it is committed as soon as it is executed
- For multiple SQL statements in a transaction:
  - BEGIN ATOMIC
    .....
    END ATOMIC

Transactions as SQL Statements

- INSERT, DELETE and UPDATE are the write operations
- Consider two concurren
  - SELECT ID,
    FROM
    WHERE salary
  - INSERT INTO
    VALUES ('11111', 'James', '
- Result changes depending on the schedule
  - Unrepeatable read can happen
- Not captured by the simple READ(X) / WRITE(X) model
  - SQL query does not explicitly say which data location is accessed

ID	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Transactions as SQL Statements

- Access of SQL query depends on WHERE clause

- Result of query can change if there are concurrent Transactions
  - No Repeatable reads
  - Solution is to consider locking the tuples that are accessed by a transaction and the information used to find the tuples (More on this later)

  - Can we know if two concurrent SQL queries access the same location?

Transactions as SQL Statements

- Consider the two queries:
  - SELECT ID,
    FROM
    WHERE salary

  - UPDATE
    SET salary = sala
    WHERE name

ID	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

- Wu is not part of result of Q1
  - If the system lock only relevant rows for Q1 (using indexing)
  - Then Q2 and Q1 are not in conflict
  - Can be decided using Predicate Locking

Transactions as SQL Statements

- Consider the two queries:
  - SELECT ID,
    FROM
    WHERE salary

  - UPDATE
    SET salary = sala
    WHERE name

ID	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

- Predicate locking
  - Treat insert, delete or update as conflicting with a predicate on a relation, if it could affect the set of tuples selected by a predicate.
  - “Salary > 90000” is a predicate. Conflicts could be
    - Updating Wu’s salary from 90000 to something greater than 90000
    - Updating Einstein’s salary from 95000 to something less or equal to 90000
  - Expensive; Not used in practice

Reference:

Database System Concepts by Silberschatz, Korth and Sudarshan
(6th edition)
Chapter 14