Part 13: Transactions

Database System Concepts, 7th Ed.

Transaction Concept

- A **transaction** is a *unit* of work that accesses and possibly updates various data items
- E.g., transaction to transfer $50 from account A to account B:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B)*
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

ACID Properties

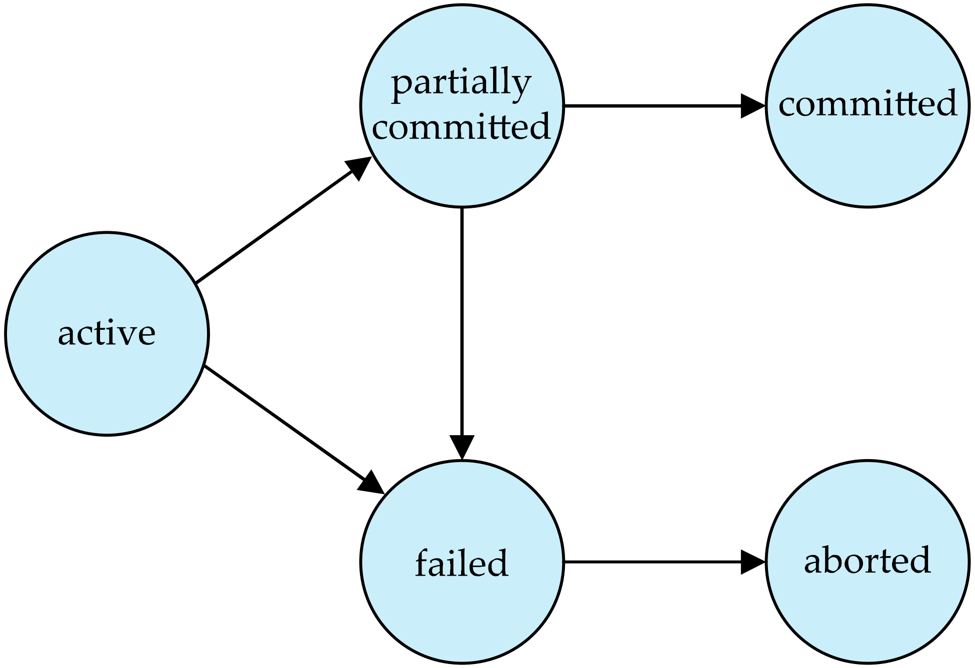
A **transaction** is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are (all or nothing)
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database
- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions
  - That is, 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

Transaction State

- **Active** – the 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 restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - Restart the transaction
    - Can be done only if no internal logical error
  - Kill the transaction
- **Committed** – after successful completion

Transaction State



Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system Advantages are:
  - **Increased processor and disk utilization**, leading to better transaction *throughput*
    - E.g., one transaction can be using the CPU while another is reading from or writing to the disk
  - **Reduced average response time** for transactions: short transactions need not wait behind long ones
- **Concurrency control schemes** – mechanisms to achieve isolation
  - That is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

Schedules

- **Schedule** – a sequences of instructions that specify 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
- A transaction that successfully completes its execution will have a commit instructions as the last statement
  - By default transaction assumed to execute commit instruction as its last step

Schedule 1

- Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance of A from A to B
- A serial schedule 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)	
commit	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit

Schedule 2

- A serial schedule where T_2 is followed by T_1

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	

Schedule 3

- 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 (i.e. T_1 followed by T_2)

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

- In Schedules 1, 2 and 3, the sum A+B is preserved

Schedule 4

- The following concurrent schedule does not preserve the value of (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)	
commit	
	$B := B + temp$
	write (B)
	commit

Serializability

- **Basic Assumption** – Each transaction is assumed correct if executed on its own
- Thus, serial execution of a set of transactions is assumed correct
- **Criterion for correctness: every serial schedule is considered correct**
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notion of **conflict serializability**

Simplified view of transactions

- We ignore operations other than **read** and **write** instructions
- We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes
- Our simplified schedules consist of only **read** and **write** instructions

Conflicting Instructions

- Instructions *I* and *J* of transactions T_i and T_j respectively, **conflict** if and only if there exists some item *Q* accessed by both *I* and *J,* and at least one of these is a write instruction
  1. *I* = **read**(*Q*)*, J* = **read**(*Q*). *I* and *J* don't conflict (the order of *I* & *J* does not matter)
  2. *I* = **read**(*Q*)*, J* = **write**(*Q*). They conflict (the order of *I* & *J* matters: write before read and read before write gives different read results)
  3. *I* = **write**(*Q*)*, J* = **read**(*Q*). They conflict (the order of *I* & *J* matters)
  4. *I* = **write**(*Q*)*, J* = **write**(*Q*). They conflict (the order of *I* & *J* matters since it would affect the result of the next **read**(*Q*) instruction)

- Intuitively, a conflict between *I* and *J* forces a (logical) temporal order between them
- If *I* and *J* are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule

Conflict Serializability

- 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**
- Or equivalently, two schedules are conflict equivalent if the relative order of any two conflicting instructions is the same in both schedules
- We say that a schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule

Conflict Serializability

- Schedule 1 can be transformed into Schedule 2, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore Schedule 1 is conflict serializable

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

Schedule 1

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

Schedule 2

Conflict Serializability

- Example of a schedule that is not conflict serializable:

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

- T_4's update is lost
- 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 >
- This is not conflict serializable since it is not equivalent to either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >

Recoverable Schedules

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

- **Recoverable schedule** — if a transaction T_j reads a data item previously written by a transaction T_i , then the commit operation of T_i appears before the commit operation of T_j.
- The following schedule is not recoverable

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

- If T_8 should abort, T_9 would have read an inconsistent database state but T_9 has already committed. Hence, database must ensure that schedules are recoverable

Cascading Rollbacks

- **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)

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		

  If T_{10} fails, T_{11} and T_{12} must also be rolled back. The read (A) in T_{11} and T_{12} is called **dirty read**

- Can lead to the undoing of a significant amount of work

Cascadeless Schedules

- **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
- Every Cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless

Weak Levels of Consistency

- Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable
  - E.g., a read-only transaction that wants to get an approximate total balance of all accounts
  - Such transactions need not be serializable with respect to other transactions
- Trade accuracy for performance

Transaction Definition in SQL

- In SQL, a transaction begins implicitly
- A transaction in SQL ends by:
  - **Commit work** commits current transaction and begins a new one
  - **Rollback work** causes current transaction to abort
- In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully

Transaction Support in SQL

- Isolation levels
  - **Dirty read** (reading of the update of an uncommitted transaction)
  - **Nonrepeatable read** (another transaction updates a data item between two reads so that the transaction sees two different values)
  - **Phantoms** (if another transaction inserts a new record *r* during the execution of the transaction, *r* was not there at the beginning of the transaction but was there at the end of the transaction; *r* is called a **phantom record**)

	Type of Violation		
Isolation Level	Dirty Read	Nonrepeatable Read	Phantom
READ UNCOMMITTED	Yes	Yes	Yes
READ COMMITTED	No	Yes	Yes
REPEATABLE READ	No	No	Yes
SERIALIZABLE	No	No	No

Possible violations based on isolation levels as defined in SQL