Chapter 14: Transactions

Database System Concepts, 6th Ed.

Transaction Concept

- A **transaction** is a *unit* of program execution 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

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

Atomicity

- "All or nothing"
- The system should ensure that updates of a partially executed transaction are not reflected in the database

- 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*)

System crash

- Money will be "lost" leading to an inconsistent database state

Consistency

- When the transaction completes successfully, the database must be consistent
  - During transaction execution, the database may be temporarily inconsistent
  - A transaction must see a consistent database
- 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)*

  - The sum of A and B is unchanged by the execution of the transaction
- In general, consistency requirements include
  - Explicitly specified integrity constraints such as primary keys and foreign keys
  - Implicit integrity constraints
    - e.g. sum of balances of all accounts, minus sum of loan amounts

Isolation

- Each transaction must be unaware of other concurrently executing transactions
  - Intermediate transaction results must be hidden from other concurrently executed transactions
  - Isolation can be ensured trivially by running transactions **serially**
    - That is, one after the other.
- Transaction to transfer $50 from account A to account B:

T1	**T2**
1. **read**(*A*)	
2. *A* := *A* – 50	
3. **write**(*A*)	
	read(A), read(B), print(A+B)
4. **read**(*B*)	
5. *B* := *B* + 50	
6. **write**(*B*)	

If T2 accesses the partially updated database, it will see an inconsistent database (the sum *A* + *B* will be less than it should be).

- However, executing multiple transactions concurrently has significant benefits, as we will see later.

Durability

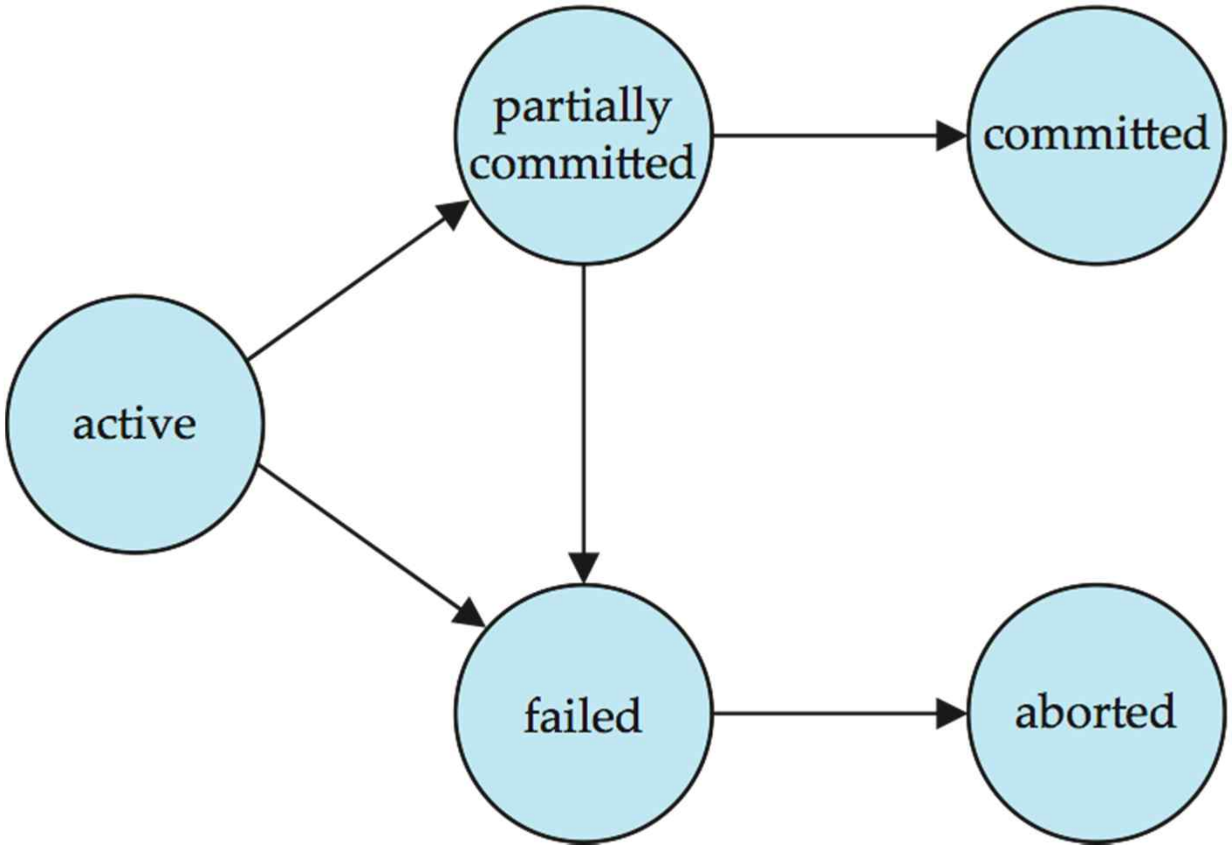
- The updates to the database by the transaction must persist even if there are software or hardware failures

- 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*)
  - Once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), it must persist

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 (Cont.)



Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **Increased processor and disk utilization**, leading to better transaction *throughput*
  - **Reduced average response time** for transactions: short transactions need not wait behind long ones

- **Concurrency control schemes** – mechanisms to achieve isolation
  - 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
- Serial schedule – instruction sequences from one by one transactions

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

Schedule 1

- Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance 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 in which 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

- The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1
  - We call it a serializable schedule

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). The following schedule is not serializable.

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 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**

Conflicting Instructions

- 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

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

Conflict Serializability

- Schedules S and S' are **conflict equivalent** if S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions
- 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 >.

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

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

Conflict Serializability (Cont.)

- Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore Schedule 3 is conflict serializable.

Schedule3

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

Schedule 5 –
After Swapping a Pair of non conflicting Instructions in schedule 3

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

Schedule 6 –
A Serial Schedule That is Equivalent to Schedule 3

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

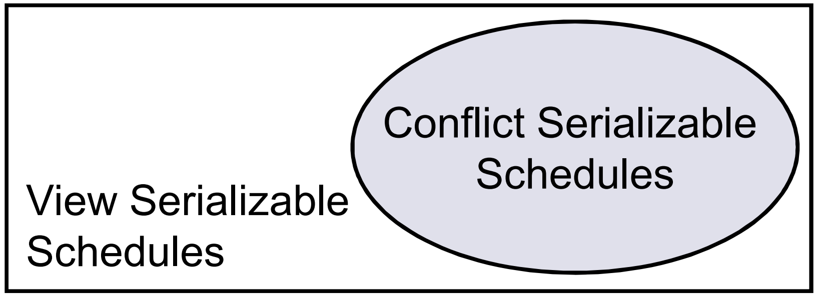
View Serializability

- Schedules S and S' are **view equivalent** if the following three conditions are met, for each data item Q:
  1. If in S, transaction T_i reads the initial value of Q, then in S' also transaction T_i must read the initial value of Q.
  2. If in S, T_i executes **read**(Q), and that value was produced by T_j (if any), then in S' also T_i must read the value of Q that was produced by the same **write**(Q) operation of T_j .
  3. The transaction (if any) that performs the final **write**(Q) operation in S must also perform the final **write**(Q) operation in S'.

- A schedule S is **view serializable** if it is view equivalent to a serial schedule

View Serializability (Cont.)

- Every conflict serializable schedule is also view serializable



- Below is a schedule which is view-serializable but *not* conflict serializable.

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

A view equivalent serial schedule

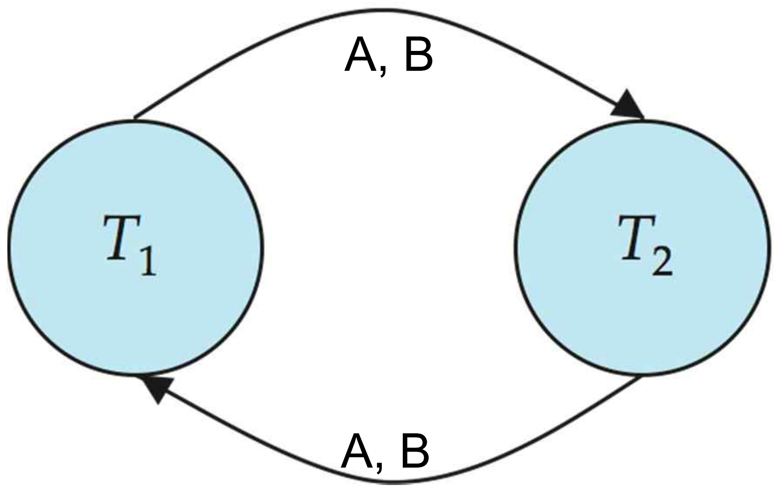
T_{27}	T_{28}	T_{29}
read (Q)		
write (Q)		
	write (Q)	
		write (Q)

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

Testing for Serializability

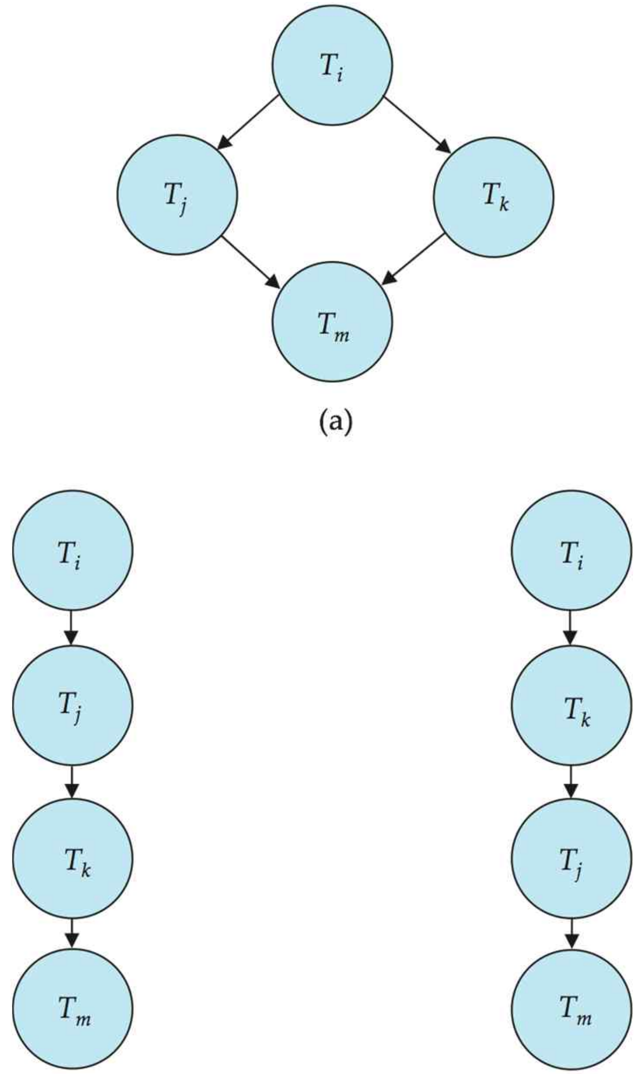
- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- **Precedence graph** – a direct graph where the vertices are transactions(names)
  - draw an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier
  - may label the arc by the item that was accessed
- Example

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)



Test for Conflict Serializability

- A schedule is conflict serializable if and only if its precedence graph is acyclic
- Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph
  - (Better algorithms take order $n + e$ where e is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph
  - A linear order consistent with the partial order of the graph



(a) (b) (c)

Precedence Graph for Serial Schedules

$$T_1 \xrightarrow{A,\ B} T_2$$

(a) Schedule 1

$$T_2 \xrightarrow{A,\ B} T_1$$

(b) Schedule 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)

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

Test for View Serializability

- The precedence graph test for conflict serializability cannot be used directly to test for view serializability
  - Extension to test for view serializability has cost exponential in the size of the precedence graph
- The problem of checking if a schedule is view serializable falls in the class of *NP*-complete problems
  - Thus existence of an efficient algorithm is *extremely* unlikely
- However, practical algorithms that just check some **sufficient conditions** for view serializability can still be used

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 if T_9 commits immediately after the read

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

- DBMS must ensure that schedules are recoverable

Cascading Rollbacks

- 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		

- 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

- Idea: block other transactions until executing the commit instruction
  - More concurrency ➔ More cascading rollback
  - Less cascading rollback ➔ Less concurrency

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serializable, and
  - are recoverable and preferably cascadeless

- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
- Testing a schedule for serializability *after* it has executed is a little too late
- **Goal** – to develop concurrency control protocols that will assure serializability
  - Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur

Weak Levels of Consistency

- Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable
  - Some transactions need not be serializable with respect to other transactions
    - E.g. a read-only transaction that wants to get an approximate total balance of all accounts
    - E.g. database statistics computed for query optimization can be approximate (why?)
- Tradeoff accuracy for performance

Levels of Consistency in SQL-92

- **Serializable** — default
- **Repeatable read** — only committed records to be read, repeated reads of same record must return same value
  - However, a transaction may not be serializable – it may find some records inserted by a transaction but not find others
- **Read committed** — only committed records can be read, but successive reads of record may return different (but committed) values
- **Read uncommitted** — even uncommitted records may be read

- Warning: some database systems do not ensure serializable schedules by default
  - E.g. Oracle and PostgreSQL by default support a level of consistency called snapshot isolation (not part of the SQL standard)

Transaction Definition in SQL

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction
- 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
  - Implicit commit can be turned off by a database directive
    - E.g. in JDBC,   connection.setAutoCommit(false);

End of Chapter 14

Database System Concepts, 6th Ed.