Chapter 14: Transactions

Outline

- Transaction Concept

- Transaction State

- Concurrent Executions

- Serializability

- Recoverability

- Implementation of Isolation

- Transaction Definition in SQL

- Testing for Serializability.

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

Required Properties of a Transaction

1. **Atomicity requirement**: Either all operations of the transaction are reflected properly in the database, or none are. Clearly, lack of atomicity will lead to inconsistency in the database.

2. **Consistency requirement**: Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database. This is typically the responsibility of the application programmer who codes the transactions.

3. **Isolation requirement**: When multiple transactions execute concurrently, it should be the case that there is no interference between transactions. Each transaction is unaware of other transactions executing concurrently with it. The property that concurrent schedules take the system from one consistent state to another.

    Even though multiple transactions may execute concurrently, 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. Thus, each transaction is unaware of other transactions executing concurrently in the system.

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

Required Properties of a Transaction

- Consider a 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)*

- **Atomicity requirement**

  - If the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state

    ▸ Failure could be due to software or hardware

  - The system should ensure that updates of a partially executed transaction are not reflected in the database

- **Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

Required Properties of a Transaction (Cont.)

- **Consistency requirement** in above example:
  - 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 must equal value of cash-in-hand
- A transaction, when starting to execute, must see a consistent database.
- During transaction execution the database may be temporarily inconsistent.
- When the transaction completes successfully the database must be consistent
  - Erroneous transaction logic can lead to inconsistency

Required Properties of a Transaction (Cont.)

■ **Isolation requirement** — if between steps 3 and 6 (of the fund transfer transaction) , another transaction **T2** is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).

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	

■ Isolation can be ensured trivially by running transactions **serially**

- That is, one after the other.

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

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.

- **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.
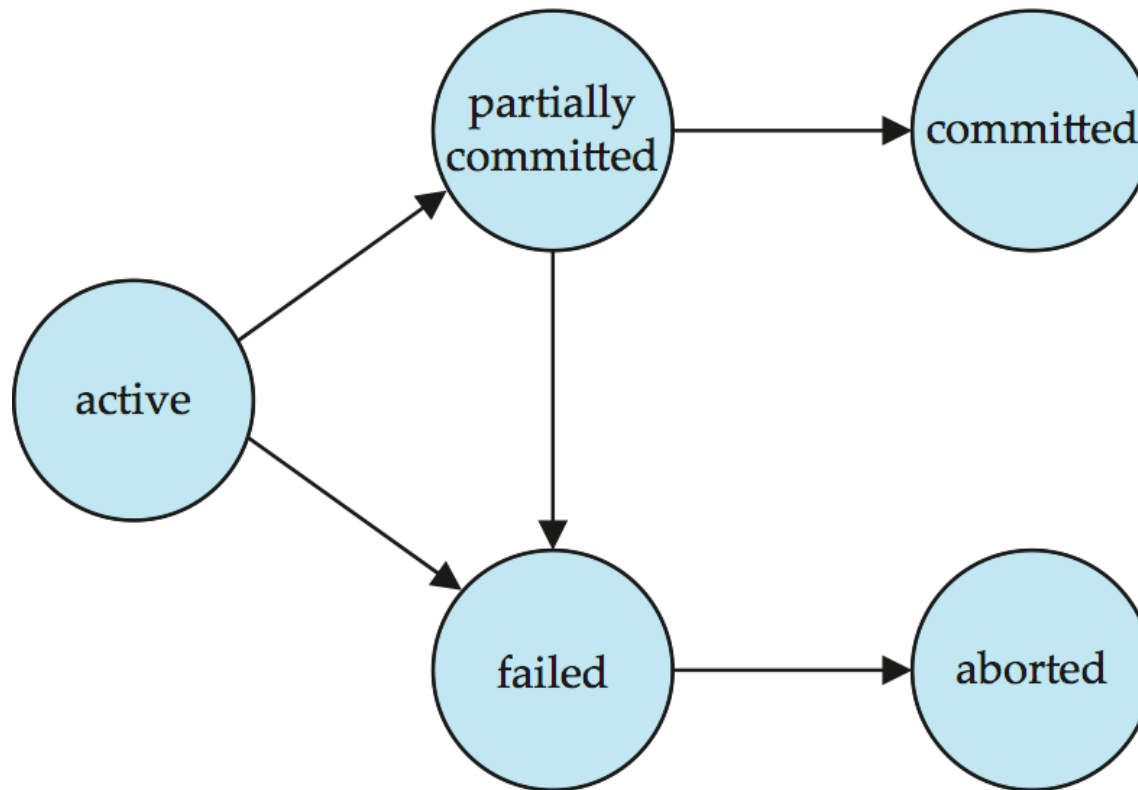
Transaction State

- **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 (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*

        - 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

        - Will study in Chapter 15, after studying notion of correctness of concurrent executions.

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

■ A transaction that fails to successfully complete its execution will have an **abort** instruction as the last statement

Schedule 1

- Let T_1 transfer $50 from A to B, and T_2 transfer 10% of the balance from A to B.
- An example of 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

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

Note -- In schedules 1, 2 and 3, the sum "A + B" is preserved.

Schedule 4

- The following concurrent schedule does not preserve the sum 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 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**

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

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

  1. $I_i = $ **read**(Q), $I_j = $ **read**(Q). I_i and I_j don't conflict.
  2. $I_i = $ **read**(Q), $I_j = $ **write**(Q). They conflict.
  3. $I_i = $ **write**(Q), $I_j = $ **read**(Q).   They conflict
  4. $I_i = $ **write**(Q), $I_j = $ **write**(Q). They conflict

- Intuitively, a conflict between I_i and I_j forces a (logical) temporal order between them.

  - If I_i and I_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**.

- We say that a schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule

Conflict Serializability (Cont.)

- Schedule 3 can be transformed into Schedule 6 -- a serial schedule where T_2 follows T_1, by a 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)

Schedule 3

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

Schedule 6

Conflict Serializability (Cont.)

■ 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 $< T_4, T_3 >$.

Precedence Graph

- Consider some schedule of a set of transactions T_1, T_2, ..., T_n

- **Precedence graph** — a direct graph where the vertices are the transactions (names).

- We 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.

- We may label the arc by the item that was accessed.

- **Example**
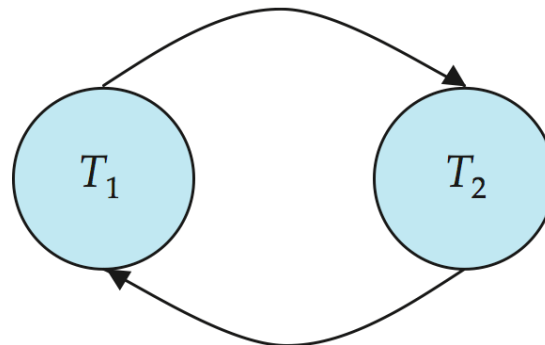
Testing for Conflict Serializability

- 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.

  - That is, a linear order consistent with the partial order of the graph.

  - For example, a serializability order for the schedule (a) would be one of either (b) or (c)
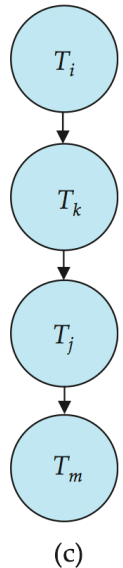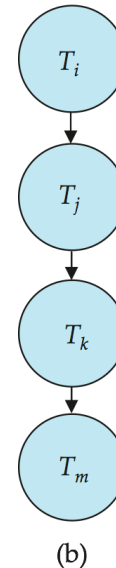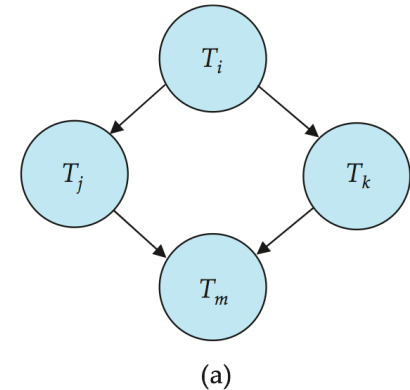
Recoverable Schedules

- **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 **must** appear before the commit operation of T_j.

- The following schedule is not recoverable if T_9 commits immediately after the read(A) operation.

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

- 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.

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.

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

Cascadeless Schedules

- **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 recoverable

- It is desirable to restrict the schedules to those that are cascadeless

- Example of a schedule that is NOT cascadeless

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

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are both:
  - Conflict serializable.
  - 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

- Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur

- Testing a schedule for serializability *after* it has executed is a little too late!
  - Tests for serializability help us understand why a concurrency control protocol is correct

- **Goal** – to develop concurrency control protocols that will assure serializability.

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

  - E.g., database statistics computed for query optimization can be approximate (why?)

  - Such transactions need not be serializable with respect to other transactions

- 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.

- Lower degrees of consistency useful for gathering approximate information about the database

- 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);

Other Notions of Serializability

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, for each data item Q,

    1. If in schedule S, transaction T_i reads the initial value of Q, then in schedule S' also transaction T_i must read the initial value of Q.

    2. If in schedule S transaction T_i executes **read**(Q), and that value was produced by transaction T_j (if any), then in schedule S' also transaction T_i must read the value of Q that was produced by the same **write**(Q) operation of transaction T_j.

    3. The transaction (if any) that performs the final **write**(Q) operation in schedule S must also 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** if it is view equivalent to a serial schedule.

- 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)

- What serial schedule is above equivalent to?

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

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.

More Complex Notions of Serializability

- The schedule below produces the same outcome as the serial schedule $< T_1, T_5 >$, yet is not conflict equivalent or view equivalent to it.

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)

- If we start with A = 1000 and B = 2000, the final result is 960 and 2040

- Determining such equivalence requires analysis of operations other than read and write.

End of Chapter 14