



Part 4: Transaction Management

Database System Concepts, 6th Ed.

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use



Transaction

- Example: Transfer of money from one account to another is a transaction consisting of two updates, one to each account
- A collection of operations that form a single logical unit of work
 - 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)**
 - If **A=\$1000** and **B=\$ 2000**
 - If Success and only T1, *then at the end $A=\$950$ and $B=\$2050$,*
 - *Thus $A+B=3000$ (**before 1000+2000** and **after 950+2050**)*



Issues

- In case of some **failure** in complete execution, partial effects of each **incomplete transaction be undone**, called **Atomicity**
- If **successful**, then its effects must persist in the database, called **Durability**
- **Multiple transactions** executing concurrently which may result in erroneous updates, require **Isolation**, to isolate transactions from others
- Chapter 14
 - Study of INDIVIDUAL transaction in detail
 - ACID Properties
 - Isolation by Serializability
- Chapter 15
 - Concurrency control techniques for **CONCURRENT** transactions Isolation
- Chapter 16
 - Recovery management for Atomicity and Durability for **CONCURRENT** transactions



Chapter 14: Transactions

Database System Concepts, 6th Ed.

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use



Chapter 14: Transactions

- INDIVIDUAL Transaction Concept
- Transaction State
- Concurrent Executions
- Serializability
- Recoverability
- Implementation of Isolation
- Transaction Definition in SQL
- Testing for Serializability



Transaction Concept

- **Transaction**

- 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)
- Main issues to deal with:
 - Completely execute
 - Concurrent execution of multiple transactions
 - Failures of various kinds, such as hardware failures and system crashes



Example of Fund Transfer

- 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)

- Suppose either of following happens

1. **read**(A)
2. $A := A - 50$
3. **write**(A)

OR

1. **read**(A)
2. $A := A - 50$
3. **write**(A)
4. **read**(B)
5. $B := B + 50$



Example of Fund Transfer

- **Atomicity requirement**
 - If the transaction FAILS after step 3 or before step 6,
 - 4 Failure could be due to software or hardware
 - 4 Money will be “LOST” leading to an INCONSISTENT database state
 - The system should ensure that updates of a partially executed transaction are not reflected in the database
 - In an atomic transaction,
 - 4 A series of database operations either ALL occur,
 - OR
 - 4 NOTHING occurs



Example of Fund Transfer (Cont.)

- **Consistency requirement**
 - The information in the database must be kept in a consistent state
 - For example:
 - 4 If we have one table *B* that includes keys from another table *A* (such as **bank account transactions** that have account IDs referencing rows in a table of **bank accounts**),
 - Then **shouldn't allow deleting a row from *A* without deleting those rows from *B*** referring to the account being deleted
 - To ensure consistency for an individual transaction is the responsibility of the **application programmer** who codes the transaction



Example of Fund Transfer (Cont.)

- **Consistency requirement**
 - Task facilitated by **automatic testing of integrity constraints**
 - **Explicitly specified integrity constraints** such as
 - 4 Primary keys and Foreign keys
 - **Implicit integrity constraints**
 - 4 e.g. sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
 - In given example, The sum of A and B is unchanged by the execution of the transaction ($A+B=3000$ (*before 1000+2000 and after 950+2050*))



Example of Fund Transfer (Cont.)

- **Consistency requirement**
 - A transaction must see a consistent database
 - During transaction execution the database may be temporarily inconsistent
 - When the transaction completes successfully the database must be consistent
- 4 Erroneous transaction logic can lead to inconsistency



Example of Fund Transfer (Cont.)

- Isolation requirement

T1

1. **read(A)**
2. $A := A - 50$
3. **write(A)**

T2

4. **read(B)**
5. $B := B + 50$
6. **write(B)**

read(A), read(B), print($A+B$)

- Computes $A+B$

- *If $A = \$1000$ and $B = \$2000$*
- *And only $T1$, then at the end $A = \$950$ and $B = \$2050$,*
- *Thus $A+B = 3000$ (before $1000+2000$ and after $950+2050$)*

- But, if $T2$ occurs, then $T2$ will observe an Inconsistent value**

- As it accesses the partially updated database, it will see an INCONSISTENT database (**the sum $A + B = \$950 + \$2000 = \$2950$ will be less than it should be**)



Example of Fund Transfer (Cont.)

- **Isolation requirement**

- If between steps 3 and 6, **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)**
4. **read(B)**
5. $B := B + 50$
6. **write(B)**

read(A), read(B), print($A+B$)

- DB is temporarily inconsistent after Step 3, *If T2 concurrently occurs and reads A and B at intermediate point*
 - Computes $A+B$, *It will observe an Inconsistent value (2950 not 3000)*
 - **Furthermore, if T2 then performs updates on A and B based on the inconsistent values that it read, the database may be left in an inconsistent state even after both transactions have completed**



Example of Fund Transfer (Cont.)

- **Isolation requirement**
 - Isolation can be ensured trivially by running transactions **serially**
 - 4 That is, one after the other
 - However, executing multiple transactions concurrently has significant benefits, as we will see later



Example of Fund Transfer

- **Durability requirement**
 - Once the user has been notified that the transaction has completed (i.e., the transfer of the \$50 has taken place),
 - 4 Then the updates to the database by the transaction must **PERSIST**
 - Even if there are DBMS Software crashes or HARDWARE FAILURES



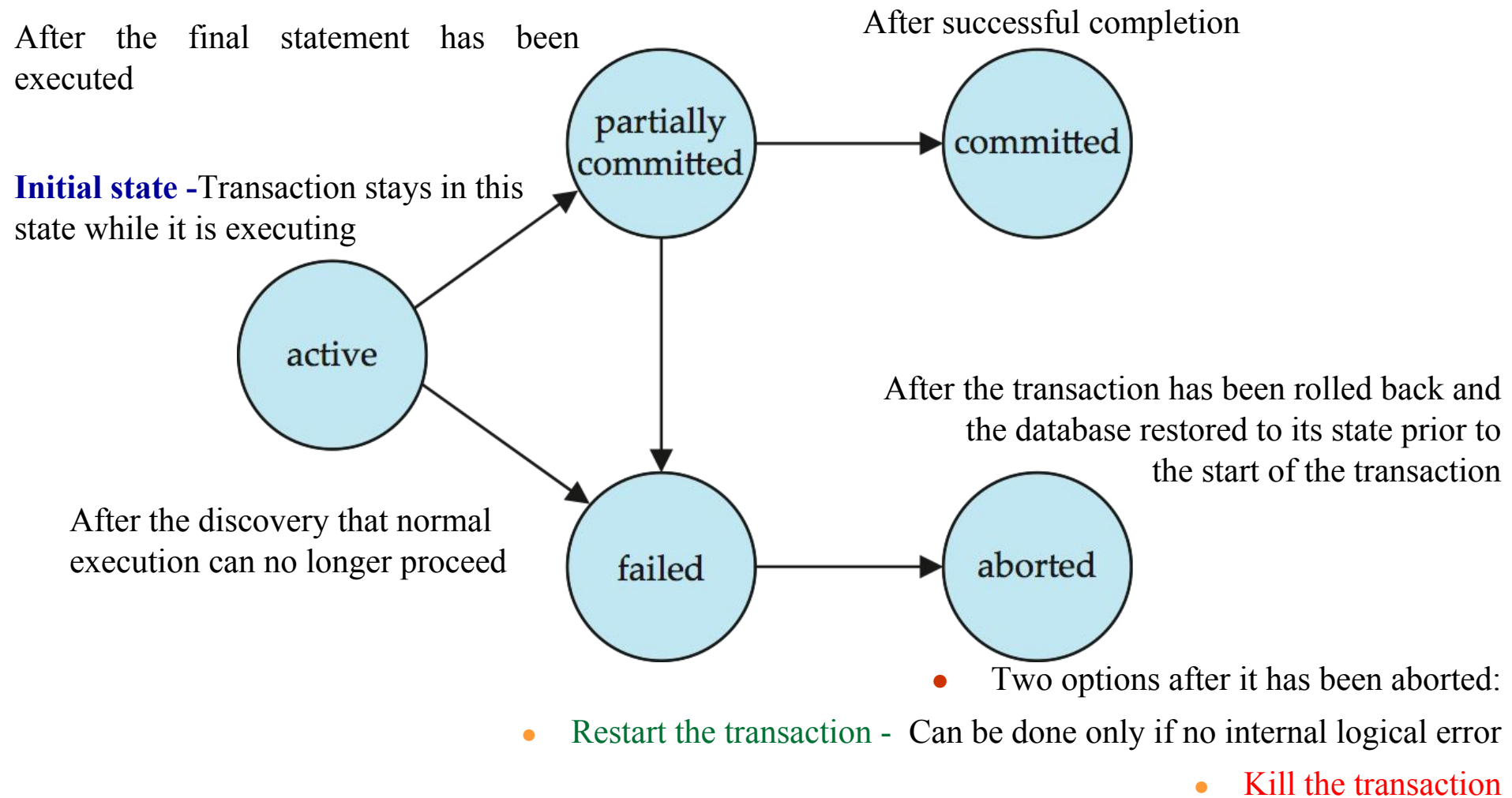
ACID Properties

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
 - 4 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 (Cont.)





Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system
- Advantages
 - **Increased processor and disk utilization**, leading to better transaction *throughput*
 - 4 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
 - To control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database
 - 4 Will study in Chapter 16, after studying notion of correctness of concurrent executions



Schedules

- **Schedule**
 - A chronological execution sequence of transaction
 - 4 A schedule can have many transactions in it,
 - Each transaction comprising of number of instructions/tasks
 - **Must preserve the order** in which the **instructions appear in each individual transaction**
- It is some interleaving of the operations from the two transactions (without violating the order of operations within any individual transaction)
- A transaction that successfully completes its execution will have a commit instructions as the last statement (By default)
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement



Example of Schedule

- Let
 - T_1 transfer \$50 from A to B , and
 - T_2 transfer 10% of the balance from A to B
- Write the possible schedules.
- Verify: $A=\$1000$, $B=\$2000$
 - Before execution of both T_1 and T_2 $\square A+B=3000$
 - After execution of both T_1 and T_2 $\square A+B=3000$
- First, write operations:

Transaction 1:

$r1(A)$, $A=A-50$, $w1(A)$, $r1(B)$, $B=B+50$, $w1(B)$,

Transaction 2:

$r2(A)$, $tmp=A*0.10$, $A=A-temp$, $w2(A)$, $r2(B)$, $B=B+temp$, $w2(B)$



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

Suppose current values of accounts
 $A = \$1000$ and $B = \$2000$

Before starting: $A+B=\$3000$

Schedule 1: T_1 followed by T_2

After T_1 : $A=\$950$, $B=\$2050$

**After T_2 : $A = \$855(=950-95)$,
 $B = \$2145(=2050+95)$**

$A + B = \$855 + \$2145 = \$3000$

Preserved after the execution of both transactions



Schedule 2

- A serial schedule where T_2 is followed by T_1

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

Suppose current values of accounts

$A = \$1000$ and $B = \$2000$

Before starting: $A+B=\$3000$

Schedule 1: T_2 followed by T_1

**After T_2 : $A=\$900(=1000-100)$,
 $B=\$2100(=2000+100)$**

After T_1 : $A = \$850, B = \2150

$A + B = \$850 + \$2150 = \$3000$

Preserved after the execution of both transactions



Serial Schedule

- T_1 and T_2 schedules are **serial**:
 - Each serial schedule consists of a sequence of instructions from various transactions, where the instructions belonging to one single transaction appear together in that schedule
 - Recalling a well-known formula from combinatorics,
 - 4 For a set of n transactions, there exist n factorial ($n!$) different valid serial schedules
- To describe consistent transaction behavior when transactions run at the same time uses transaction isolation model called **serializability**
- The serializable mode of transaction behavior tries to ensure that transactions run in such a way that they appear to be executed one at a time, or serially, rather than concurrently



Schedule 3

- Let T_1 and T_2 be the transactions defined previously
- 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

Current values of accounts

$A = \$1000$ and $B = \$2000$

Before starting: $A+B=\$3000$

After Part1 of T_1 : $A=950$, $B=2000$

**After Part 1 of T_2 : $A = 855(=950-95)$,
 $B = 2000$, $temp=95$**

After Part2 of T_1 : $A=855$, $B=2050$

**After Part 2 of T_2 : $A = 855$,
 $B = 2145(=2050+95)$**

$A + B = \$855 + \$2145 = \$3000$

Preserved after the execution of both transactions



Schedule 4

Check the following concurrent schedule, is it preserving 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

Suppose the current values of accounts
 $A = \$1000$ and $B = \$2000$

$T1: A=950$

$T2: A=1000$ (not written by $T1$)

$temp=100$

$A=900$

$B=2000$

$T1: A=950$

$B=2000$

$B=2050$

$B=2000+100=2100$

Final values after execution $A = \$950, B = \2100

Total amount of money in accounts A and B , the sum $A + B$ ($= 950 + 2100 = 3050$) (Before execution = $\$3000$) is NOT preserved after the execution of both transactions

Final state is an *inconsistent state*, since we have gained \$50

and the sum $A + B$ is not preserved by the execution of the two transactions



Serial Schedule

- When the database system **executes several transactions concurrently**, the corresponding schedule no longer needs to be serial
 - If two transactions are running concurrently, the **OS** may execute one transaction for a little while, then perform a context switch, execute the second transaction for some time, and then switch back to the first transaction for some time, and so on
 - With multiple transactions, the CPU time is shared among all the transactions
 - In general, it is **not possible to predict exactly how many instructions of a transaction will be executed** before the CPU switches to another transaction



Serial Schedule

- If control of concurrent execution is left entirely to the **OS**, many possible schedules, including ones that leave the database in an inconsistent state
- **It is the job of the database system to ensure that any schedule that is executed will leave the database in a consistent state**
- **The concurrency-control component of the database system carries out this task**



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. **View serializability (Later Ch. 15 Concurrency)**
 2. **Conflict serializability**



Serializability

Conflict serializability

- Schedule is defined by equivalence to a serial schedule (no overlapping transactions) with the same transactions, **such that both schedules have the same sets of respective chronologically ordered pairs of conflicting operations (same precedence relations of respective conflicting operations)**
- Widely utilized
 - 4 Because it is easier to determine and covers a substantial portion of the view-serializable schedules



Simplified view of transactions

- Since transactions are programs, it is difficult to determine exactly
 - What operations a transaction performs and
 - How operations of various transactions interact
- For this reason, here NOT considered the various types of operations that a transaction can perform on a data item,
 - Ignore operations other than **read** and **write** instructions
- 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
- **Commit operations, though relevant, are not considered until Section 14.7**



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. $l_i = \mathbf{read}(Q)$, $l_j = \mathbf{read}(Q)$. l_i and l_j don't conflict.
 2. $l_i = \mathbf{read}(Q)$, $l_j = \mathbf{write}(Q)$. They conflict.
 3. $l_i = \mathbf{write}(Q)$, $l_j = \mathbf{read}(Q)$. They conflict
 4. $l_i = \mathbf{write}(Q)$, $l_j = \mathbf{write}(Q)$. They conflict
- Intuitively, a conflict between l_i and l_j forces a (logical) temporal order between them
 - If l_i and l_j 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**
- A schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule



Conflict Serializability (Cont.)

- Check Schedule 3, is it a conflict serializable schedule.
- Sol: Try the possibility of swaps of non-conflicting instructions, to produce serial schedule.

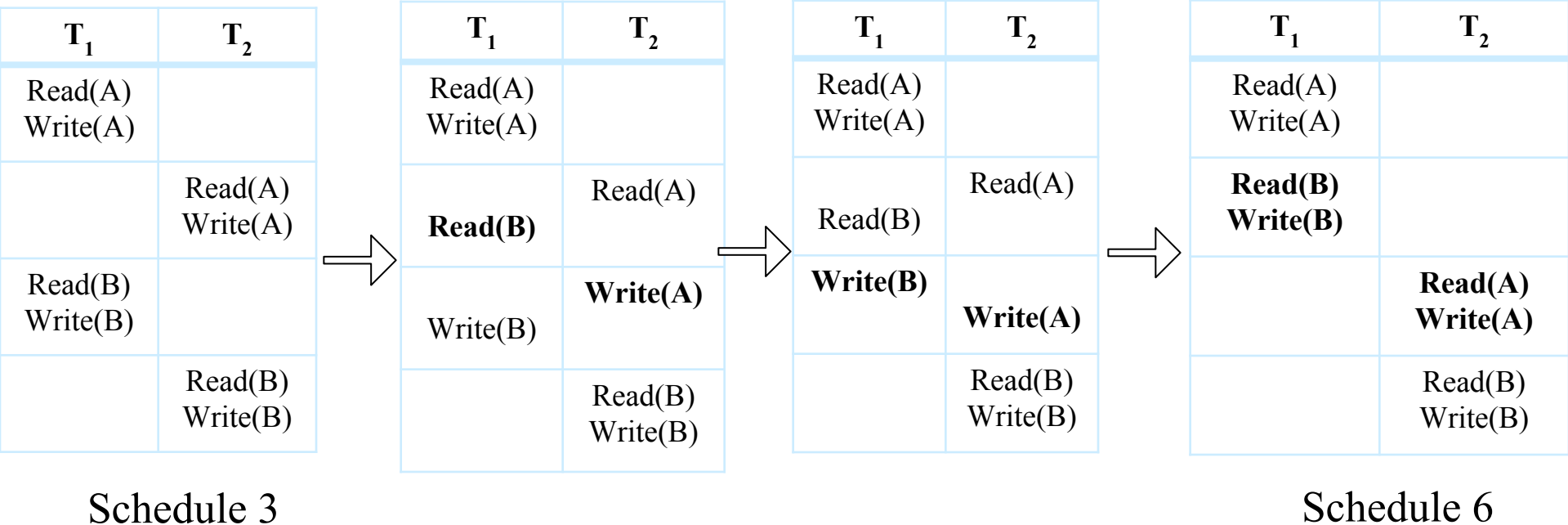
T_1	T_2
read (A) write (A)	read (A) write (A)
read (B) write (B)	read (B) write (B)

Schedule 3



Conflict Serializability (Cont.)

- Check Schedule 3, is it a conflict serializable schedule.
- Sol: Try the possibility of swaps of non-conflicting instructions, to produce serial schedule.





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**

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.)

- Check the serializability of following schedule.

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

- Schedule is **not conflict serializable**
 - Unable to swap instructions in the above schedule to obtain either the serial schedule $\langle T_3, T_4 \rangle$, or the serial schedule $\langle T_4, T_3 \rangle$
 - As read and write both are on single item Q



Testing for Serializability

- Consider some schedule of a set of transactions T_1, T_2, \dots, T_n
- **Precedence graph**
 - Simple and Efficient method to determine conflict serializability of schedule
 - A directed graph where the Vertices are the transactions (names)
 - Draw an Edge(arc) from T_i to T_j
 - 4 If the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier
 - 4 Label the arc by the item that was accessed
 - 4 Draw edge from $T_i \rightarrow T_j$ with label Q for one of three conditions:

T_i

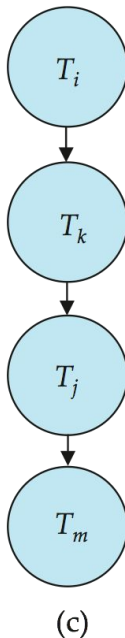
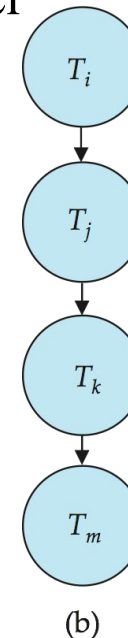
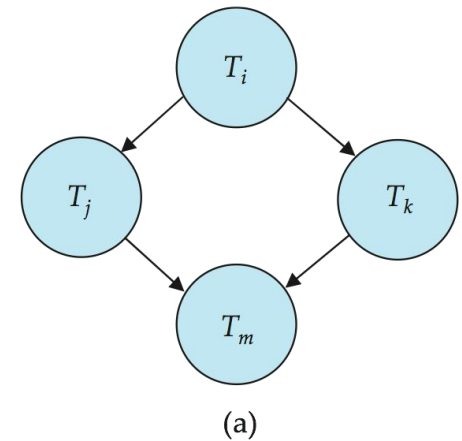
T_j

 1. $write(Q) \quad read(Q)$
 2. $read(Q) \quad write(Q)$
 3. $write(Q) \quad write(Q)$



Test for Conflict Serializability

- A schedule is **conflict serializable**
 - If and only if 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
 - This is a linear order consistent with the partial order of the graph





Testing for Serializability

- Draw edge from $T_i \rightarrow T_j$ for one of three conditions:

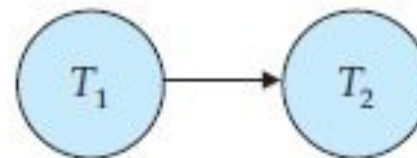
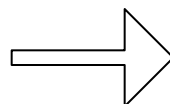
T_i T_j

1. $write(Q)$ $read(Q)$
2. $read(Q)$ $write(Q)$
3. $write(Q)$ $write(Q)$

- **Example: Schedule 1**

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

As all the instructions of T_1 are executed before the T_2



CONFLICT SERIALIZABLE Schedule



Testing for Serializability

- Draw edge from $T_i \rightarrow T_j$ for one of three conditions:

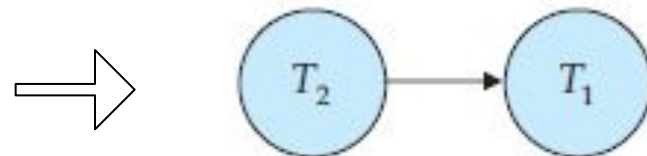
T_i T_j

- $write(Q)$ $read(Q)$
- $read(Q)$ $write(Q)$
- $write(Q)$ $write(Q)$

- Example: Schedule 2**

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

As all the instructions of T_2 are executed before the T_1



CONFLICT SERIALIZABLE Schedule

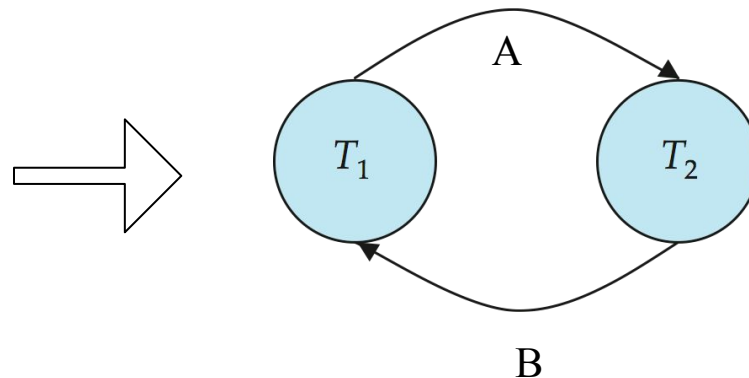


Testing for Serializability

- **Example: Schedule 4**
- Contains the edge $T_1 \rightarrow T_2$, because T_1 executes $read(A)$ before T_2 executes $write(A)$
- Also contains the edge $T_2 \rightarrow T_1$, because T_2 executes $read(B)$ before T_1 executes $write(B)$

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

NOT CONFLICT SERIALIZABLE Schedule



Cycle is present in graph



Testing for Conflict Serializability

Example: Schedule A

W3 (A)

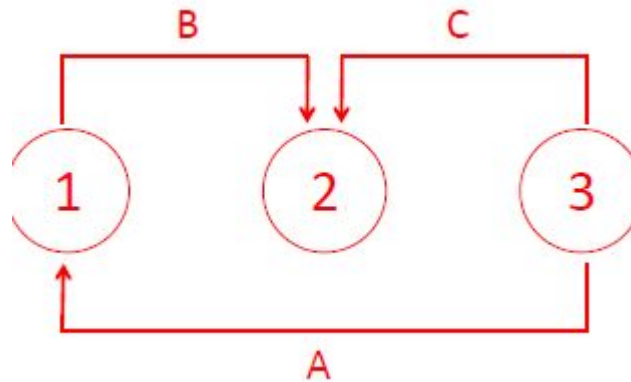
R1 (A)

W1 (B)

R2 (B)

W3 (C)

R2 (C)



No cycles: **YES, conflict serializable**



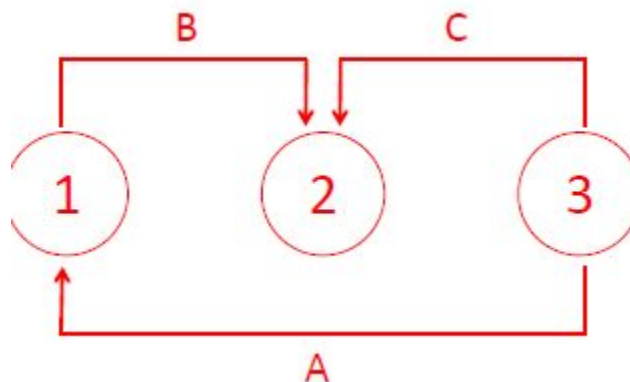
Test for Conflict Serializability

Example: Schedule A

W3 (A)
R1 (A)
W1 (B)
R2 (B)
W3(C)
R2 (C)

Serial Equivalent Schedule?

W3 (A)
W3(C)
R1 (A)
W1 (B)
R2 (B)
R2 (C)



No cycles: **YES, conflict serializable**

Only serial equivalent schedule: T3, T1, T2

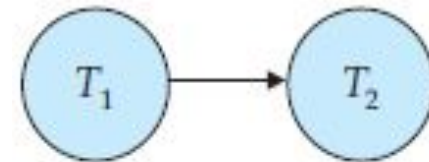
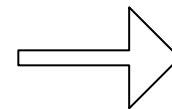


Test for Conflict Serializability

- **Example: Schedule 3**

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

As all the instructions of T_1 are executed before the completion of T_2



CONFLICT SERIALIZABLE Schedule



Transaction Failure

- Upto this considered that **NO TRANSACTION FAILURE**
 - But, this is not really true

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

- **IF TRANSACTION FAILURE** is there,
 - Need to **UNDO the effect of that transaction** to ensure **ATOMICITY**
 - Requires that any transaction T_j that dependent on T_i is also aborted
 - Achieved by restricted type of schedules

4 Recoverable Schedule and Cascadeless Schedule



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 appears before the commit operation of T_j

- The Schedule 11

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

- Only commit is with T_9 not with T_8 (**Partial Schedule**)
- T_9 reads the value written by T_8 (**T_9 is dependent upon T_8**)
- If T_8 is failed, Requires Undo for T_9 also, which is not possible



Recoverable Schedules

Schedule 11

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

- Database must ensure that schedules are recoverable
 - T_9 is **dependent** on T_8
 - Because of this, must abort T_9 to ensure atomicity
 - However, T_9 has already committed and cannot be aborted
 - » Thus, a situation where it is impossible to recover correctly from the failure of T_8
 - If T_8 should abort, T_9 would have read (and possibly shown to the user) an inconsistent database state
 - Schedule is **NOT RECOVERABLE**



Recoverable Schedules

Schedule 10

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

- Database must ensure that schedules are recoverable
 - T_{11} is dependent upon T_{10} and T_{12} is dependent on T_{11}
 - Abort is there with T_{10} , it is possible to abort T_{11} and T_{12}
 - Schedule is **RECOVERABLE**



Cascading Rollbacks

- The schedule is **RECOVERABLE, BUT REQUIRES CASCADING ROLLBACKS**
 - 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

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

- A single transaction failure leads to a series of transaction rollbacks
 - As none of the transactions has committed till abort of T_{10}



Cascadeless Schedules

- Cascading rollback is undesirable
 - As requires a lot many rollback
- For each pair of transactions T_x and T_y such that T_y reads a data item previously written by T_x , the commit operation of T_x appears before the read operation of T_y

T_x T_y

Write(Q)

Commit

Read(Q)

- Every CASCADELESS schedule is RECOVERABLE schedule



Concurrency Control

- If only one transaction is allowed to execute at a time
 - It generates serial schedules
 - BUT, provides a poor degree of concurrency
- For concurrent transactions, DBMS need to check
 - Either conflict or view serializable schedule ?
 - Recoverable and preferably cascadeless schedule ?
 - Are serial schedules recoverable/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
 - 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
 - Allowing not to wait for other transactions serial order
 - 4 E.g. Permit the transaction to read a data item even if it was written by a transaction that has not been committed
 - To benefit the long transactions whose results do not need to be precise



Transaction Isolation Levels of Consistency

- With different possible outcomes for the same transaction scenario
 - The same work performed in the same fashion with the same inputs may result in different answers, depending on your isolation level
 - Isolation levels
 - 4 **Serializable (Default)**
 - 4 **Repeatable read**
 - 4 **Read committed**
 - 4 **Read uncommitted**
 - These levels are defined in terms of three phenomena that are either permitted or not permitted at a given isolation level



Transaction Isolation Levels of Consistency

- **Serializable (Default)**
 - Serializable schedule are the ideal way to ensure consistency, but in our day-to-day lives, NOT IMPOSING SUCH STRINGENT REQUIREMENT
 - Ensured serializability only for the transactions that run on the database, without user interaction
 - In Online Shopping, when user surfing for an item it is available in stock, but by the time user goes through the checkout process, that item might no longer be available □ this would be a nonrepeatable read
- **Repeatable read**
- **Read committed**
- **Read uncommitted**



Transaction Isolation Levels of Consistency

- **Serializable (Default)**
 - Ensured serializability only for the transactions that run on the database, without user interaction
- **Repeatable read**
 - Allows only committed data to be read and further requires that, between two reads of a data item by a transaction, **no other transaction is allowed to update it**
 - Only committed records to be read, repeated reads of same record must **return same value**
 - However, a transaction may not be serializable with respect to other transactions, so it may find some records inserted by a transaction but not find others
- 4 For instance, when it is searching for data satisfying some conditions, a transaction may find some of the data inserted by a committed transaction, but may not find other data inserted by the same transaction (until it is committed-as this will read only committed one)
- **Read committed**
- **Read uncommitted**



Transaction Isolation Levels of Consistency

- **Serializable (Default)**
- **Repeatable read**
- **Read committed**
 - Allows only committed data to be read, but successive reads of record may return different (but committed) values
 - 4 Does not require repeatable reads
 - For instance, between two reads of a data item by the transaction, another transaction **may have updated the data item and committed**
- **Read uncommitted**
 - Allows uncommitted data to be read and the lowest isolation level allowed by SQL



Transaction Isolation Levels of Consistency

Read phenomena that violate serializability

- Three different *read phenomena* when Transaction 1 reads data that Transaction 2 might have changed
 1. **Dirty read**
 2. **Nonrepeatable read**
 3. **Phantom read**
- Examples, two transactions queries use the data table: USERS

id	name	age
1	Joe	20
2	Jill	25

1. First Query 1 is performed
2. Then, in the second transaction, Query 2 is performed on same table
3. Finally, in the first transaction, Query 1 is performed again



Transaction Isolation Levels of Consistency

Read phenomena - Dirty read - Aka *uncommitted dependency*

- Occurs when a transaction is allowed to read data from a row that has been **modified by another** running transaction and **not yet committed**

id	name	age
1	Joe	20
2	Jill	25

Transaction 1

```
/* Query 1 */ SELECT age FROM users  
WHERE id = 1; /* will read 20 */
```

```
/* Query 1 */ SELECT age FROM users  
WHERE id = 1;
```

Reads the uncommitted data
will read 21 for id=1

Changes (already read by Transaction 1) or updates different changes to the database, then the view of the data may be wrong in the records of Transaction 1

Transaction 2

Changes a row,
but does not commit the changes

```
/* Query 2 */ UPDATE users SET age = 21  
WHERE id = 1; /* No commit here */
```

ROLLBACK;



Transaction Isolation Levels of Consistency

- **Read phenomena - Non-repeatable reads**
 - Occurs, when during the course of a transaction, a row is retrieved twice and the values within the row differ between reads

id	name	age
1	Joe	20
2	Jill	25

Transaction 1

```
/* Query 1 */ SELECT age FROM users  
WHERE id = 1; /* will read 20 */
```

```
/* Query 1 */ SELECT age FROM users  
WHERE id = 1; will read 21  
COMMIT;
```

Transaction 2

Changes a row, with commit

```
/* Query 2 */ UPDATE users SET age = 21  
WHERE id = 1; /* No commit here */  
COMMIT;
```

Transaction 1 has already seen a different value for *age* in that row – as non-repeatable reads



Transaction Isolation Levels of Consistency

- **Read phenomena - Phantom reads**
 - Occurs when, in the course of a transaction, two identical queries are executed, and the collection of rows returned by the second query is different from the first
 - A special case of *Non-repeatable reads* when Transaction 1 repeats a ranged *SELECT ... WHERE* query and, between both operations, Transaction 2 creates (i.e. INSERT) new rows (in the target table) which fulfill that *WHERE* clause

		id	name	age
Transaction 1	Transaction 2	1	Joe	20
		2	Jill	25
/* Query 1 */ SELECT * FROM users WHERE age BETWEEN 10 AND 30;				
		/* Query 2 */ INSERT INTO users(id,name,age) VALUES (3, 'Bob', 27); COMMIT ;		
/* Query 1 */ SELECT * FROM users WHERE age BETWEEN 10 AND 30; COMMIT ;				

A different set of rows may be returned the second time



Transaction Isolation Levels of Consistency

- **Dirty read**
 - The meaning of this term is as bad as it sounds
 - Related to reading UNCOMMITTED data, and occur when an UPDATE, INSERT, or DELETE from another transaction is read, and the other transaction has NOT yet committed the data
 - Permitted to read uncommitted or dirty data
 - It is reading "in progress" data, which may not be complete, and may never actually be committed
 - Example: Open an OS file that someone else is writing and reading whatever data happens to be there
 - Data integrity is compromised, foreign keys are violated, and unique constraints are ignored



Transaction Isolation Levels of Consistency

- **Non-repeatable and phantom read**
 - Deal with data modification operations from a different transaction, which were committed after your transaction began, and then read by your transaction
- **Nonrepeatable read**
 - Simply means that if you read a row at time T1 and try to reread that row at time T2, the row may have changed
 - Transaction reads committed **UPDATES** from another transaction
 - It may have disappeared, it may have been updated, and so on
 - The same row now has different values than it did when your transaction began



Transaction Isolation Levels of Consistency

- **Non-repeatable and phantom read**
 - Deal with data modification operations from a different transaction, which were committed after your transaction began, and then read by your transaction
- **Phantom read**
 - Similar to non-repeatable, but when reading from committed **INSERTS** and/or **DELETES** from another transaction
 - Means that if you execute a query at time T1 and re-execute it at time T2, additional rows may have been added to the database, which may affect your results
 - Differs from a non-repeatable read in that with a phantom read, data you already read hasn't been changed, but instead, more data satisfies your query criteria than before
 - 4 There are new rows or rows that have disappeared since you began the transaction



Transaction Isolation Levels of Consistency

Isolation Levels vs Read Phenomena

Isolation level	Dirty reads	Non-repeatable reads	Phantom reads
Read Uncommitted	may occur	may occur	may occur
Read Committed	-	may occur	may occur
Repeatable Read	-	-	may occur
Serializable	-	-	-



Transaction Isolation Levels of Consistency

- Explicitly you can set the isolation level
 - SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
 - SET TRANSACTION ISOLATION LEVEL READ COMMITTED
 - SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
 - SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
 - Oracle supports first two only and one more READ ONLY
 - To change, this setting has to be the first statement of a transaction
 - Also Turn OFF “Auto Commit”



Concurrency Control

- Implementation of Isolation level
 - Upto this, seen properties of schedule to leave the database in consistent state and allow transaction failures to be handled in a safe manner
 - Concurrency control policies
 - 4 When multiple transactions executed concurrently, only acceptable schedules are generated, regardless of how the OS time shares resources (such as CPU time) among the transactions



Concurrency Control

- Implementation of Isolation level
 - Upto this, seen properties of schedule to leave the database in consistent state and allow transaction failures to be handled in a safe manner
 - Concurrency control policies
 - 4 Example: Transaction acquires a lock on the entire database before it starts and releases the lock after it has committed
 - While a transaction holds a lock, no other transaction is allowed to acquire the lock, and all must therefore wait for the lock to be released
 - » Only one transaction can execute at a time, Serial schedule which is recoverable and cascadeless
 - » POOR PERFORMANCE, only one transaction, No concurrency



Concurrency Control

- Implementation of Isolation level
 - To provide high degree of concurrency with assurance of generated schedules are conflict or view serializable, recoverable and cascadeless

4 Ch. 15 Concurrency Control



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
 - 4 E.g. in JDBC, `connection.setAutoCommit(false);`



ACID Properties

To Achieve ACID Properties

- **Atomicity.** Recovery
- **Consistency.** Developer
- **Isolation.** Concurrency Controller
- **Durability.** Recovery



End of Chapter 14

Database System Concepts, 6th Ed.

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use