

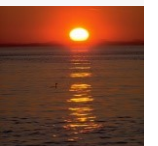


Module 6

Transaction Management, Concurrency Control and
Recovery Protocols

Database System Concepts, 5th Ed.

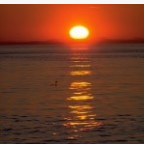
©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use





Chapter 15: Transactions

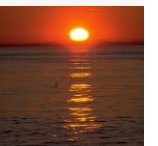
- Transaction Concept
 - Transaction State
 - Concurrent Executions
 - Serializability
 - Recoverability
 - Implementation of Isolation
 - Transaction Definition in SQL
 - Testing for Serializability.
-
- **Transaction:**
 - set of operations used to perform a logical unit of work.





Transaction Concept

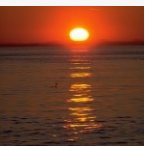
- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- E.g. transaction to transfer \$50 from account A to account B:
 1. **read**(A)
 2. $A := A - 50$
 3. **write**(A)
 4. **read**(B)
 5. $B := B + 50$
 6. **write**(B)
- Two main issues to deal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





Example of Fund Transfer

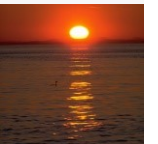
- Transaction to transfer \$50 from account A to account B:
 1. **read**(A)
 2. $A := A - 50$
 3. **write**(A)
 4. **read**(B)
 5. $B := B + 50$
 6. **write**(B)
- **Atomicity requirement**
 - if the transaction fails after step 3 and before step 6, money will be “lost” leading to an inconsistent database state
 - 4 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.





Example of Fund Transfer (Cont.)

- 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)
- **Consistency requirement** in above example:
 - the sum of A and B is unchanged by the execution of the transaction
- In general, consistency requirements include
 - 4 Explicitly specified integrity constraints such as primary keys and foreign keys
 - 4 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 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** — 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).

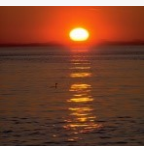
T1

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

T2

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

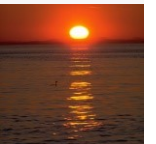




ACID Properties

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:

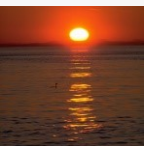
- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation.** Although multiple transactions may execute concurrently, each transaction 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.





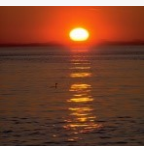
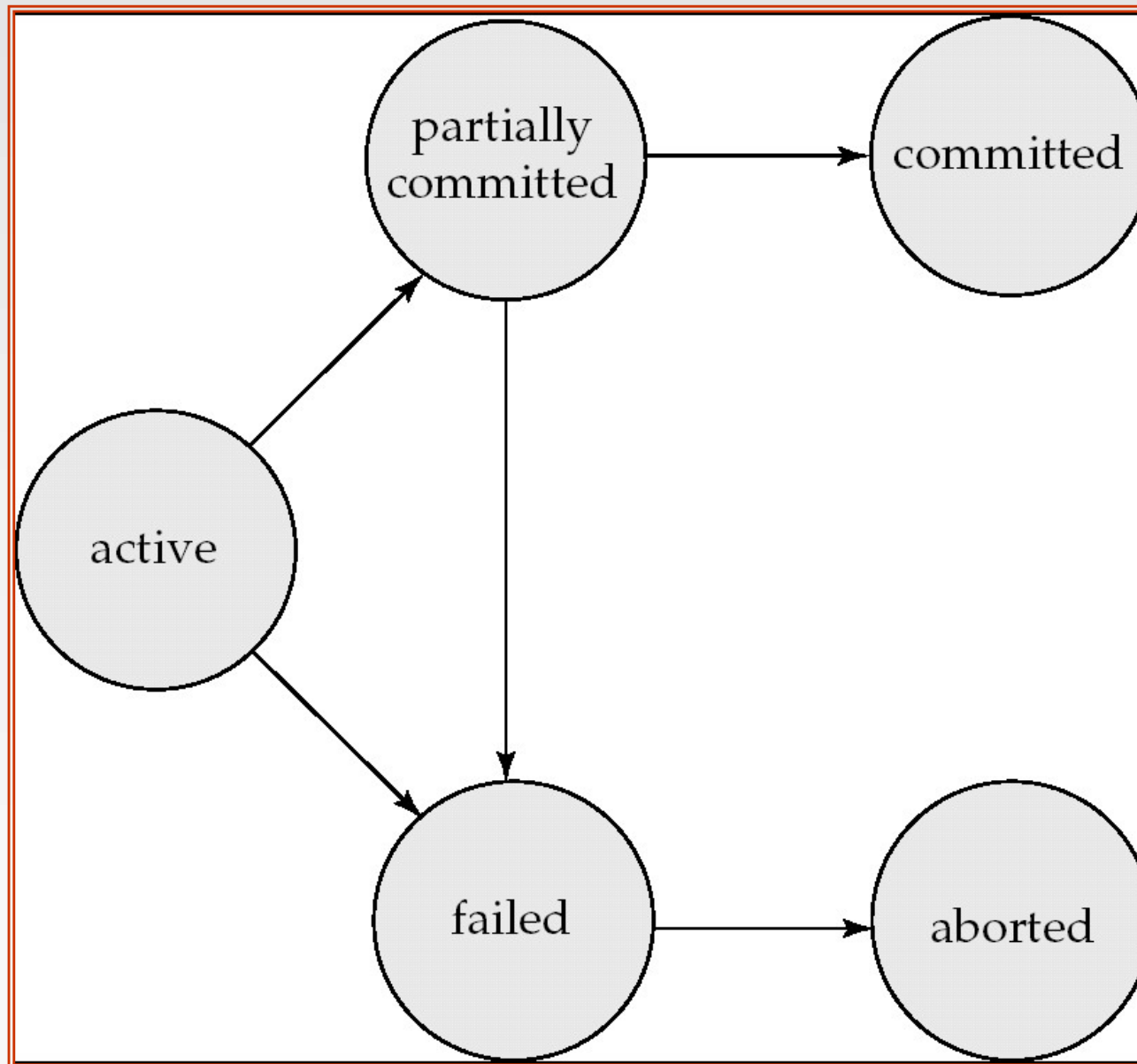
Transaction State

- **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
 - 4 can be done only if no internal logical error
 - kill the transaction
- **Committed** – after successful completion.





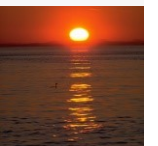
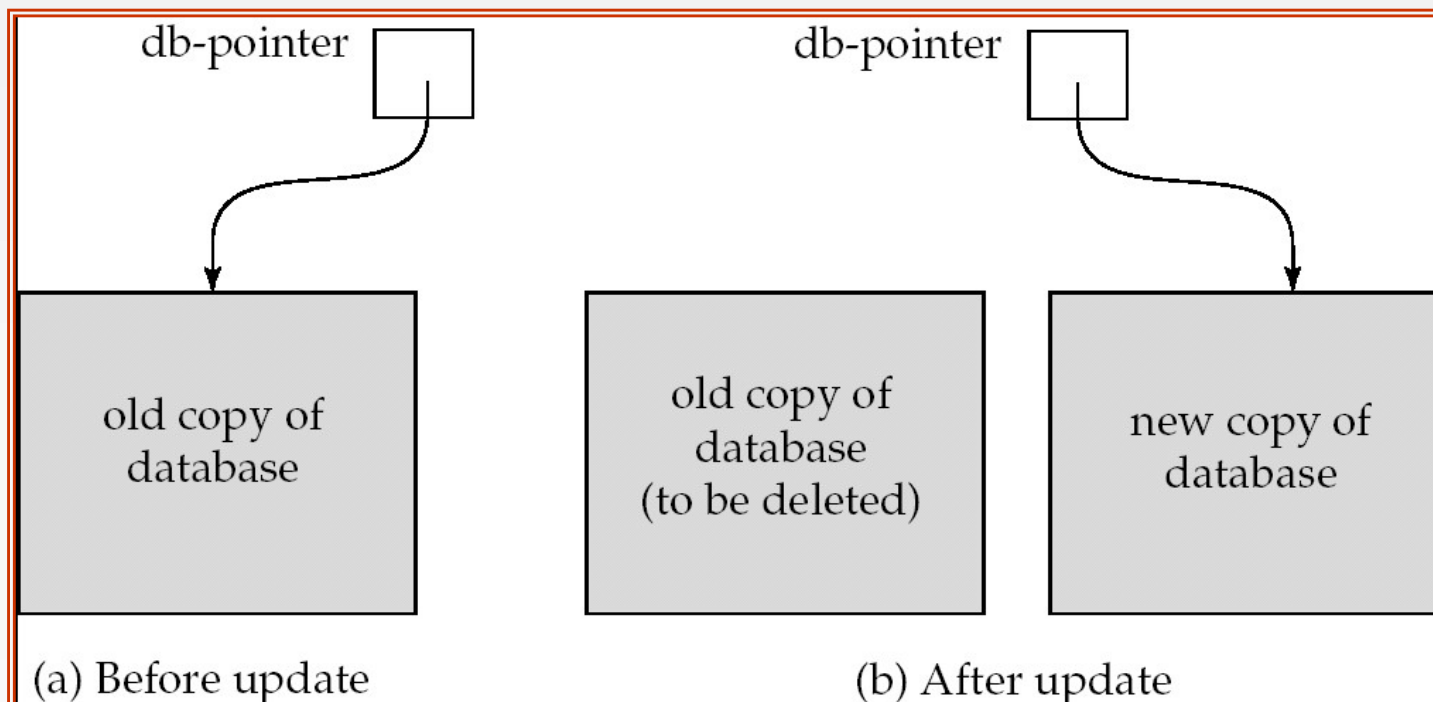
Transaction State (Cont.)





Implementation of Atomicity and Durability

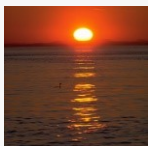
- The **recovery-management** component of a database system implements the support for atomicity and durability.
- E.g. the **shadow-database** scheme:
 - all updates are made on a *shadow copy* of the database
 - 4 **db_pointer** is made to point to the updated shadow copy after
 - the transaction reaches partial commit and
 - all updated pages have been flushed to disk.





Implementation of Atomicity and Durability (Cont.)

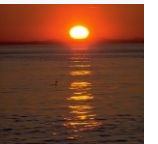
- db_pointer always points to the current consistent copy of the database.
 - In case transaction fails, old consistent copy pointed to by **db_pointer** can be used, and the shadow copy can be deleted.
- The shadow-database scheme:
 - Assumes that only one transaction is active at a time.
 - Assumes disks do not fail
 - Useful for text editors, but
 - 4 extremely inefficient for large databases (why?)
 - Variant called shadow paging reduces performance efficiency(takes more time) by copying of data, but is still not practical for large databases
 - Does not handle concurrent transactions





Concurrent Executions

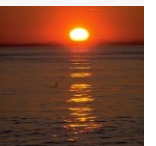
- Multiple transactions are allowed to run concurrently in the system. Advantages are:
 - **increased processor and disk utilization**, leading to better transaction *throughput*
 - 4 E.g. one transaction can be using the CPU while another is reading from or writing to the disk
 - **reduced average response time** for transactions: short transactions need not wait behind long ones.
- **Concurrency control schemes** – mechanisms to achieve isolation
 - 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
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement

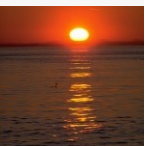




Schedule 1

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





- | 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) |



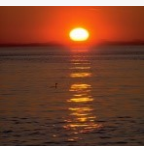


Schedule 3

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

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





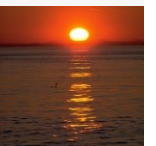
- | T_1 | T_2 |
|---|---|
| $\text{read}(A)$
$A := A - 50$ | $\text{read}(A)$
$\text{temp} := A * 0.1$
$A := A - \text{temp}$
$\text{write}(A)$
$\text{read}(B)$ |
| $\text{write}(A)$
$\text{read}(B)$
$B := B + 50$
$\text{write}(B)$ | $B := B + \text{temp}$
$\text{write}(B)$ |





Serializability

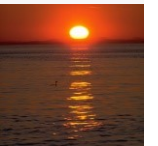
- **Basic Assumption** – Each transaction preserves database consistency.
- Thus serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to 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

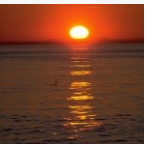
- Instructions I_i and I_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both I_i and I_j , and at least one of these instructions wrote Q .
 1. $I_i = \text{read}(Q)$, $I_j = \text{read}(Q)$. I_i and I_j don't conflict.
 2. $I_i = \text{read}(Q)$, $I_j = \text{write}(Q)$. They conflict.
 3. $I_i = \text{write}(Q)$, $I_j = \text{read}(Q)$. They conflict
 4. $I_i = \text{write}(Q)$, $I_j = \text{write}(Q)$. 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 series of swaps of non-conflicting instructions.
 - Therefore Schedule 3 is conflict serializable.
 - T_3 & T_6 are conflict serializable

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

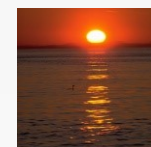
Schedule

3

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

Schedule

6



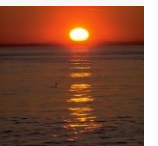


Conflict Serializability (Cont.)

- Example of a schedule that is not conflict serializable:

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

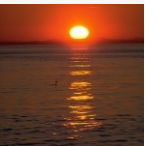
- We are unable to swap instructions in the above schedule to obtain either the serial schedule $\langle T_3, T_4 \rangle$, or the serial schedule $\langle T_4, T_3 \rangle$.





View Serializability

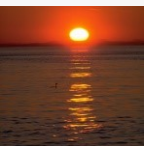
- Let S and S' be two schedules with the same set of transactions. S and S' are **view equivalent** if the following three conditions are met, for each data item Q ,
 - If in schedule S , transaction T_i reads the initial value of Q , then in schedule S' also transaction T_i must read the initial value of Q .**
 - 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' .**





3. If in schedule S transaction T_i executes $\text{read}(Q)$, 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 $\text{write}(Q)$ operation of transaction T_j .

As can be seen, view equivalence is also based purely on **reads** and **writes** alone.





View Serializability (Cont.)

- Schedule x can be transformed into Schedule y, a serial schedule where T_2 follows T_1 , by series of swaps of instructions.

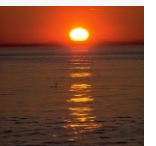
T_1	T_2
read(A) write(A)	read(A) write(A)
read(B) write(B)	
	read(B) write(B)

Schedule x

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

Schedule y

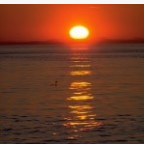
- A schedule X is view equivalent to a serial schedule Y





View Serializability (Cont.)

- A schedule S is **view serializable** if it is view equivalent to a serial schedule.
- So schedule X is **view serializable**, since it is view equivalent to a serial schedule Y
- Every conflict serializable schedule is also view serializable.

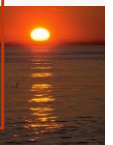




View Serializability (Cont.)

- what serial schedule is above equivalent to?
- Below is a schedule which is view-serializable but *not* conflict serializable.
- Every view serializable schedule that is not conflict serializable has **blind writes**.
- The write operations which are not read later are called blind writes

T_3	T_4	T_6
read(Q)	write(Q)	
write(Q)		
		write(Q)



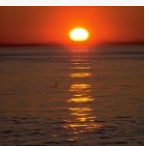


Other Notions of Serializability

- The schedule below produces same outcome as the serial schedule $\langle T_1, T_5 \rangle$, 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)

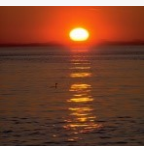
- Determining such equivalence requires analysis of operations other than read and write.





Testing of Serializability in DBMS with Examples

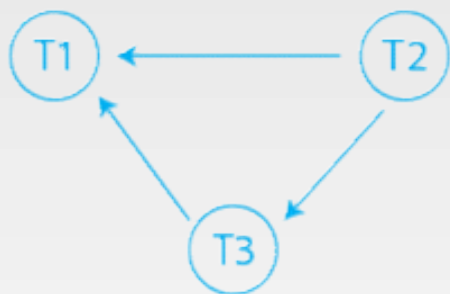
- Serializability testing involves verifying that a given schedule of transactions is serializable, meaning that the effects of running the transactions concurrently are equivalent to running them serially, one after the other.
- We can use below two techniques to test serializability in DBMS:
- Serialization Graph
- Precedence Graph
- It can be described as a Graph $G(V, E)$ with vertices $V = "V1, V2, V3, \dots, Vn"$ and directed edges $E = "E1, E2, E3, \dots, En"$. One of the two operations—READ or WRITE—performed by a certain transaction is contained in the collection of edges.
- The meaning of the following graph is that, before Transaction y , Transaction x is performing a read or write operation.





Conflict serializability example

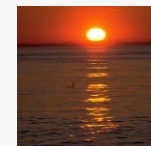
- Let's see how a precedence graph looks like of above schedule.



- In the above precedence graph,
- we can see that there is no cycle present in the graph.

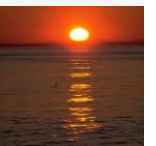
So, we can say that the above schedule is **conflict serializable**.

Transaction 1	Transaction 2	Transaction 3
R(x)		
		R(y)
		R(x)
	R(y)	
	R(z)	
		W(y)
	W(z)	
R(z)		
W(x)		
W(z)		





- As there is no loop in the graph, it is a conflict serializable schedule as well as a serial schedule.
- Since it is a serial schedule, we can detect the order of transactions as well.
- **The order of the Transactions:** t1 will execute first as there is no incoming edge on T1. t3 will execute second as it depends on T1 only. t2 will execute at last as it depends on both T1 and T3.
- Possibility of shedule: T1->T2->T3, T1->T3->T2, T2->T1->T3, T2->T3->T1, T3->T1->T2, T3->T2->T1
- So, order of its equivalent serial schedule is:
- Count the degree of each node: (incoming arrow) if it is zero consider 1st schedule,
- Final schedule order is T2-> T3->T1
- <https://www.scaler.com/topics/dbms/serializability-in-dbms>



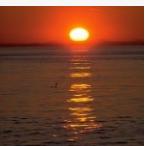


Example of View Serializability

- Let's swap the read-write operations in the middle of the two transactions to create the view equivalent schedule

Transaction 1	Transaction 2
R(a)	
W(a)	
	R(a)
	W(a)
R(b)	
W(b)	
	R(b)
	W(b)

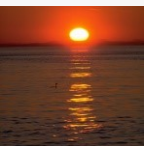
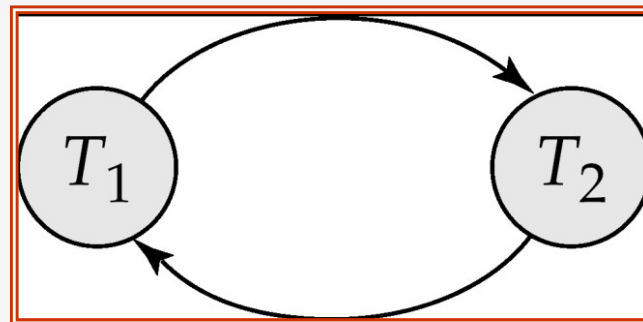
Transaction 1	Transaction 2
R(a)	
W(a)	
R(b)	
W(b)	
	R(a)
	W(a)
	R(b)
	W(b)





Testing for Serializability

- Consider some schedule of a set of transactions T_1, T_2, \dots, 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 1**





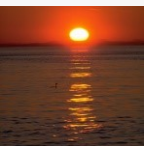
Recoverable Schedules

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 (Schedule 11) is not recoverable if T_9 commits immediately after the read

T_8	T_9
read(A)	read(A)
write(A)	
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.





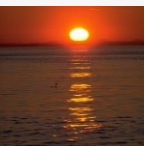
Cascading Rollbacks

- **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)

If T_{10} fails, T_{11} and T_{12} must also be rolled back.

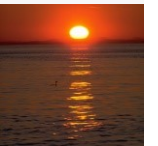
- Can lead to the undoing of a significant amount of work





Cascadeless Schedules

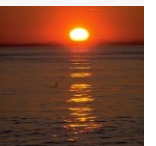
- **Cascadeless schedules** — cascading rollbacks cannot occur; for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i , the commit operation of T_i appears before the read operation of T_j .
- Every cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless





Concurrency Control

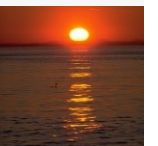
- A database must provide a mechanism that will ensure that all possible schedules are
 - either conflict or view serializable, and
 - are recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
 - Are serial schedules recoverable/cascadeless?
- 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 vs. Serializability Tests

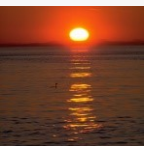
- Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless .
- Concurrency control protocols generally do not examine the precedence graph as it is being created
 - Instead a protocol imposes a discipline that avoids nonserializable schedules.
 - We study such protocols in Chapter 16.
- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.
- Tests for serializability help us understand why a concurrency control protocol is correct.





Weak Levels of Consistency

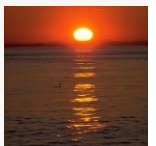
- Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable
 - E.g. a read-only transaction that wants to get an approximate total balance of all accounts
 - E.g. database statistics computed for query optimization can be approximate (why?)
 - 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
 - 4 E.g. in JDBC, `connection.setAutoCommit(false);`

