Chapters 14-16: Transaction Management

- Transactions (Chapter 14)
  - Transaction Concept
  - Transaction State
  - Concurrent Executions
  - Serialisability
  - Recoverability
  - Testing for Serialisability
- Concurrency control (Chapter 15)
  - Lock-based protocols
  - Timestamp-based protocols
  - Multiple granularity
  - Multiversion schemes
- Recovery Systems (Chapter 16)
  - Log-based recovery
  - Recovery with concurrent transactions
- Transaction in SQL
- Transaction management in Oracle

Concept of Transaction

- 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_from_account**(*A*)
  2. *A* := *A* – 50
  3. **write_to_account**(*A*)
  4. **read_from_accont**(*B*)
  5. *B* := *B* + 50
  6. **write_to_account**(*B*)
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

Transaction ACID properties

- E.g. transaction to transfer €50 from account A to account B:
  1. **read_from_acoount**(*A*)
  2. $A := A - 50$
  3. **write_to_account**(*A*)
  4. **read_from_accont**(*B*)
  5. $B := B + 50$
  6. **write_to_account**(*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
  - **All or nothing**, regarding the execution of the transaction
- **Durability requirement** — once the user has been notified of the transaction's completion, the updates must persist in the database even if there are software or hardware failures.

Transaction ACID properties (Cont.)

- Transaction to transfer €50 from account A to account B:
  1. **read_from_acoount**(*A*)
  2. $A := A - 50$
  3. **write_to_account**(*A*)
  4. **read_from_accont**(*B*)
  5. $B := B + 50$
  6. **write_to_account**(*B)*
- **Consistency requirement** in the 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 must see a consistent database and must leave a consistent database
- During transaction execution the database may be temporarily inconsistent.
  - Constraints are to be verified only at the end of the transaction

Transaction ACID properties (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)	
	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 - Summary

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 (single) transaction 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.

Non-ACID Transactions

- There are application domains where ACID properties are not necessarily desired or, most likely, not always possible.
- This is the case of so-called **long-duration transactions**
  - Suppose that a transaction takes a lot of time
  - In this case it is unlikely that isolation can/should be guaranteed
    - E.g. Consider a transaction of booking a hotel and a flight
- Without Isolation, Atomicity may be compromised
- Consistency and Durability should be preserved

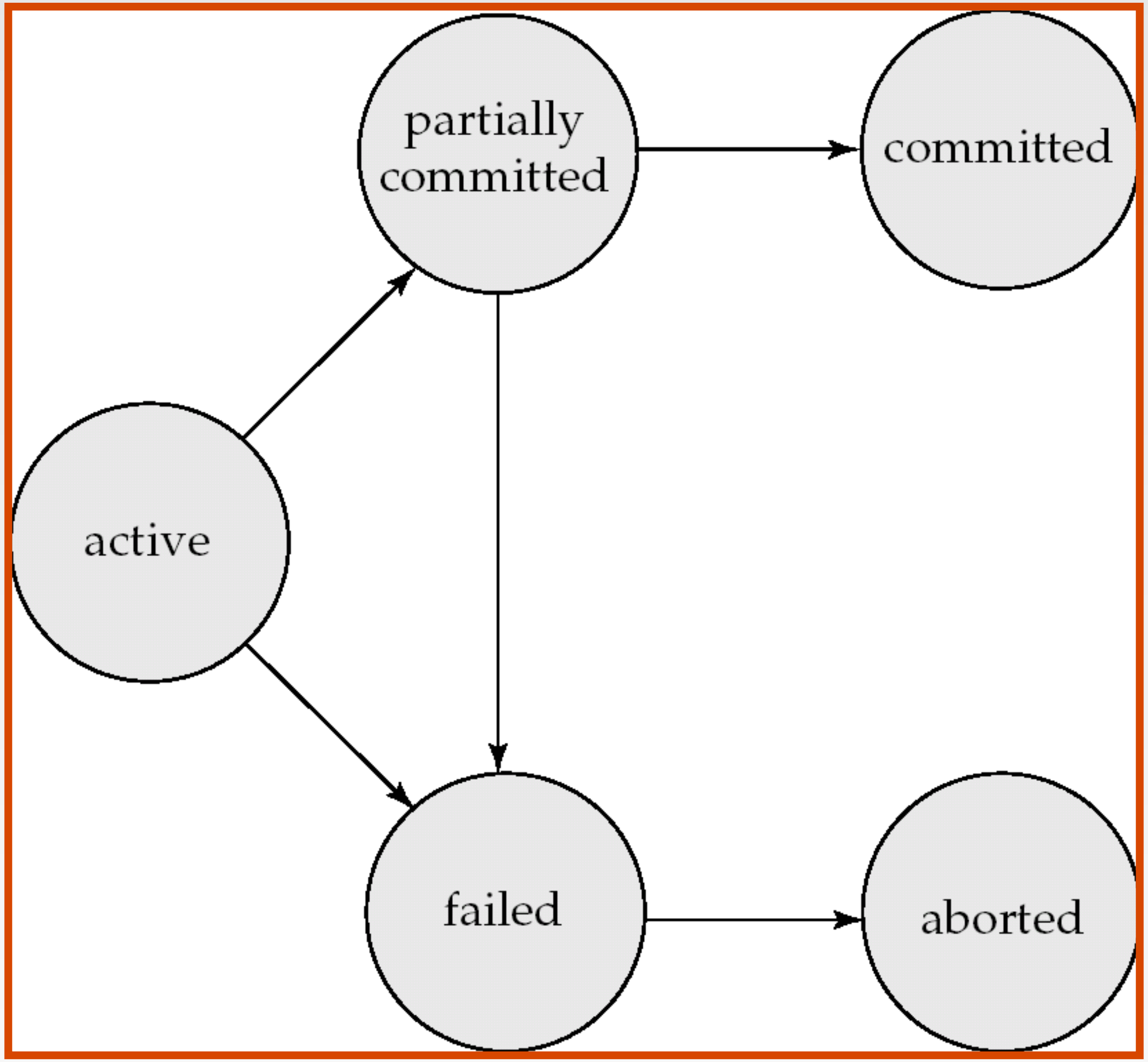
- A usual solution for long-duration transactions is to define **compensation actions** – what to do if later the transaction fails
- In (centralised) databases long-duration transactions are usually not considered.
- But these are more and more important, especially in the context of the Web.

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.

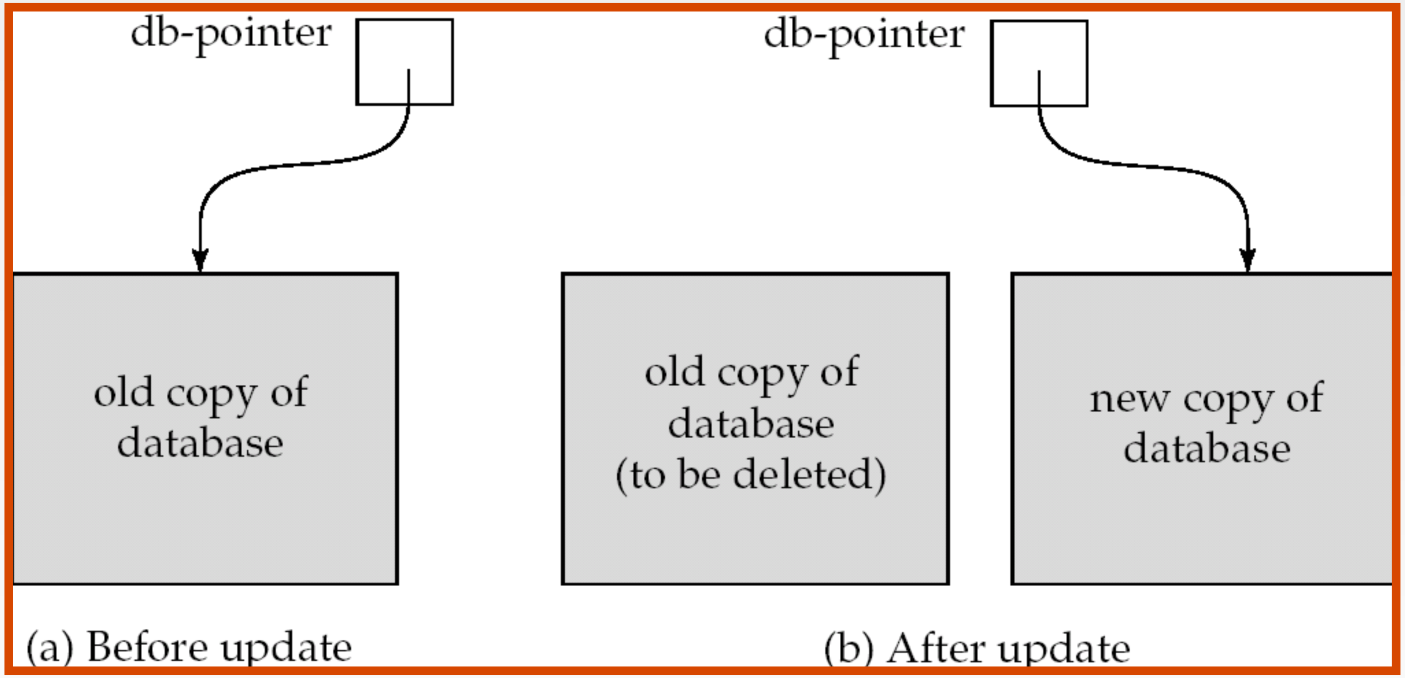
- To guarantee atomicity, external observable actions should all be performed (in order) after the transaction is committed.

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
    - **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.
  - If the 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 extremely inefficient for large databases(!)
    - Variant called shadow paging reduces copying of data, but is still not practical for large databases
  - Does not handle concurrent transactions
- Other implementations of atomicity and durability are possible, e.g. by using logs.
  - Log-based recovery will be addressed later.

Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **increased processor and disk utilisation**, 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
    - Two-phase look protocol
    - Timestamp-Based Protocols
    - Validation-Based Protocols
  - Studied in Operating Systems, and briefly summarised later

Schedules

- **Schedule** – a sequences of instructions that specifies 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, the transactions shown here are 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

- The goal is to find schedules that preserve the consistency.

Example 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)

Example Schedule 2

- A serial schedule where T_2 is followed by T_1

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

Example 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.

Example Schedule 4

– The following concurrent schedule does not preserve the value of ($A + B$).

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

Serialisability

- **Goal :** Deal with concurrent schedules that are equivalent to some serial execution:
  - **Basic Assumption** – Each transaction preserves database consistency.
  - Thus serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serialisable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:
  1. **conflict serialisability**
  2. **view serialisability**
- *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 = **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 an 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 Serialisability

- 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 serialisable** if it is conflict equivalent to a serial schedule
- 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 it is conflict serialisable.

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 Serialisability (Cont.)

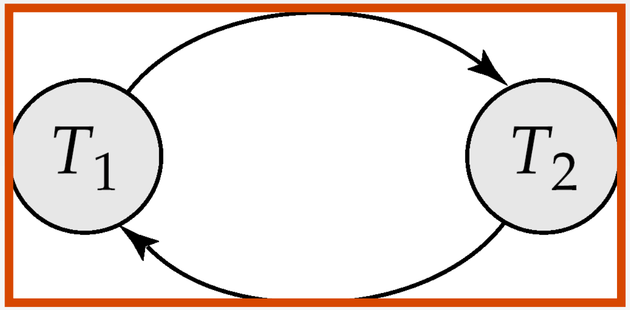
– Example of a schedule that is not conflict serialisable:

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 $< T_4, T_3 >$.

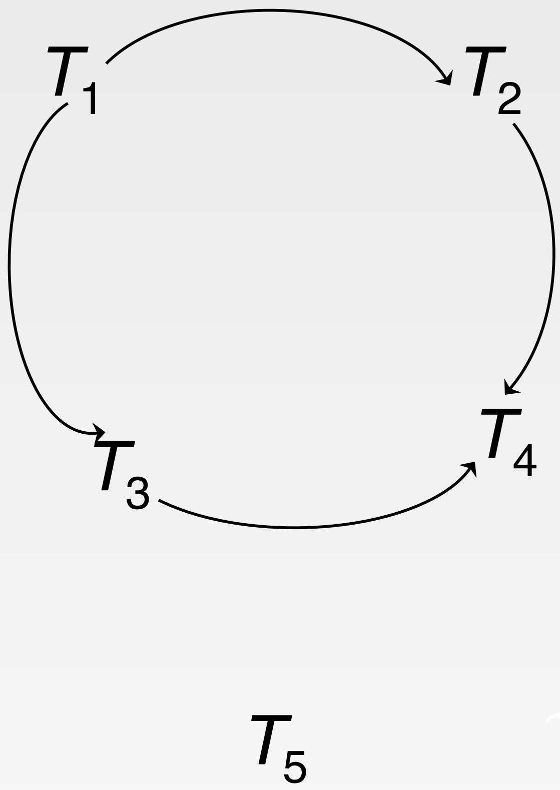
Testing for Serialisability

- 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).
  - there is 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**



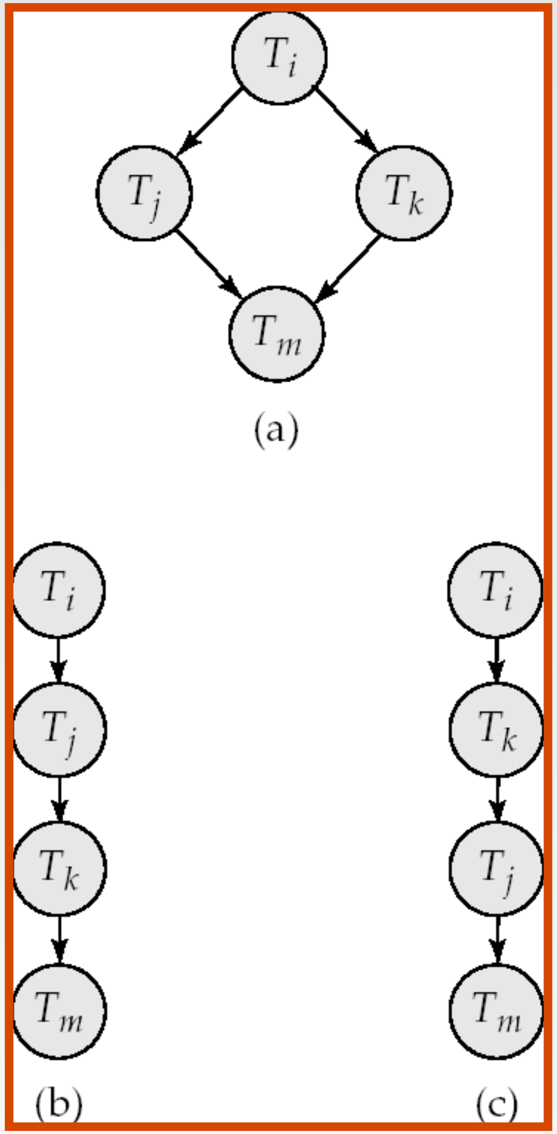
Example Schedule (Schedule A) + Precedence Graph

T_1	T_2	T_3	T_4	T_5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				read(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				
write(U)				



Test for Conflict Serialisability

- A schedule is conflict serialisable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which take $O(n^2)$ 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 the precedence graph is acyclic, the serialisability order can be obtained by a *topological sorting* of the graph.
  - I.e. a linear order consistent with the partial order of the graph.
  - E.g. a serialisability order for Schedule A would be
    $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$



View Serialisability

- Sometimes it is possible to serialise schedules that are not conflict serialisable

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

- This schedule is not conflict serialisable
- But it is serialisable:
  - It is equivalent to either <T3,T4,T6> or <T4,T3,T6>
- **View serialisability** provides a weaker and still consistency preserving notion of serialisation

View Equivalence

- 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'.*
- A schedule *S* is **view serialisable** if it is view equivalent to a serial schedule.
  - Every conflict serialisable schedule is also view serialisable
  - Every view serialisable schedule that is not conflict serialisable has **blind writes.**

Test for View Serialisability

- The precedence graph test for conflict serialisability cannot be used directly to test for view serialisability.
  - Extension to test for view serialisability has cost exponential in the size of the precedence graph.
- The problem of checking if a schedule is view serialisable 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 serialisability can still be used.

Recoverable Schedules

What to do if some transaction fails? One needs to address the effect of failures on concurrently running transactions.

- **Recoverable schedule** — if a transaction T_1 reads a data item previously written by a transaction T_2, then the commit operation of T_2 must appear before the commit operation of T_1.
- The following schedule is not recoverable if T_9 commits immediately after the read

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

- If T_8 should abort, T_9 would have read (and possibly shown to the user, or to other transactions) an inconsistent database state. Hence, a database must ensure that schedules are recoverable - *delaying commits*.

Cascading Rollbacks

- **Cascading rollback** – when 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
- Avoided in this case, by *anticipating* the commit of T_{10} to before the read in T_{11}, and the commit of T_{11} to before the read in T_{12}

Cascadeless Schedules

- **Cascadeless schedules** — in these, cascading rollbacks cannot occur; for each pair of transactions T_1 and T_2 such that T_1 reads a data item previously written by T_2, the commit operation of T_2 must appear before the read operation of T_1.
  - I.e. only committed value can be read
- 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 ensuring that all possible executed schedules are
  - either conflict or view serialisable, 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 serialisability *after* it has executed is already too late!
- **Goal** – to develop concurrency control protocols that will ensure serialisability
  - Lock-based protocols
  - Timestamp-based protocols

Concurrency Control vs. Serialisability Tests

- Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serialisable, 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 non-serialisable schedules
- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.
- Tests for serialisability help us understand why a concurrency control protocol is correct.

Optimistic vs Pessimistic protocols

T1	T2
read(A)	
	write(A)
read(B) / write(A)	
write(B)	
	read(A)

- What to do now?
  - It may well be that the complete transactions are serialisable
  - But they may also turn out not to be serialisable
- **Optimistic protocols** do not stop at potential conflicts; if something goes wrong, rollback!
- **Pessimistic protocols** stop at potential conflicts, until no possible conflict exists; if in the end no conflict happened, it just lost time!
- Let's start with a pessimistic protocol.

Lock-Based Protocols

- A lock is a mechanism to control concurrent access to a data item
- Data items can be locked in two modes :
  1. *exclusive (X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
  2. *shared (S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction.
- Lock requests are made to concurrency-control manager. A transaction can proceed only after the request is granted.

Lock-Based Protocols (Cont.)

- Lock-compatibility matrix

	S	X
S	true	false
X	false	false

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
- Any number of transactions can hold shared locks on an item,
  - but if any transaction holds an exclusive lock on the item no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made to wait until all incompatible locks held by other transactions have been released. The lock is then granted.

Lock-Based Protocols (Cont.)

- Example of a transaction performing locking:

  T_2: **lock-S**(*A*);
  read (*A*);
  unlock(*A*);
  lock-S(*B*);
  read (*B*);
  unlock(*B*);
  display(*A+B*)

- Locking as above is not sufficient to guarantee serialisability — if *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum would be wrong.
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

The Two-Phase Locking Protocol

- This is a protocol which ensures conflict-serialisable schedules.
- Phase 1: Growing Phase
  - transaction may obtain locks
  - transaction may not release locks
- Phase 2: Shrinking Phase
  - transaction may release locks
  - transaction may not obtain locks
- The protocol assures serialisability. It can be proved that the transactions can be serialised in the order of their **lock points** (i.e. the point where a transaction acquired its final lock).

Pitfalls of Lock-Based Protocols

- Consider the partial schedule

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

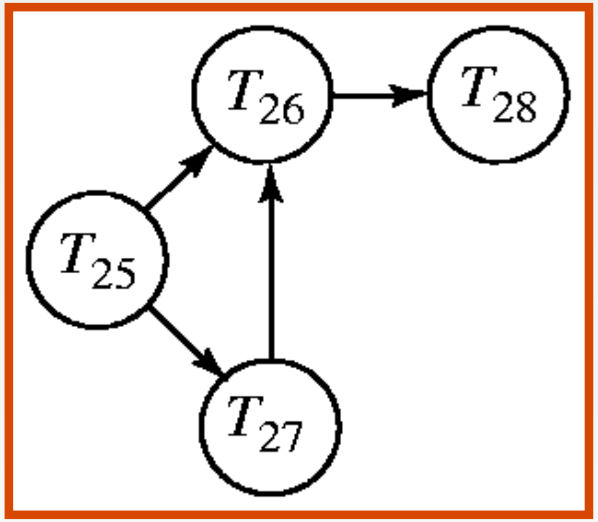
- Neither T_3 nor T_4 can make progress — executing **lock-S**(*B*) causes T_4 to wait for T_3 to release its lock on *B*, while executing **lock-X**(*A*) causes T_3 to wait for T_4 to release its lock on *A*.
- Such a situation is called a **deadlock**.
  - To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.

Pitfalls of Lock-Based Protocols (Cont.)

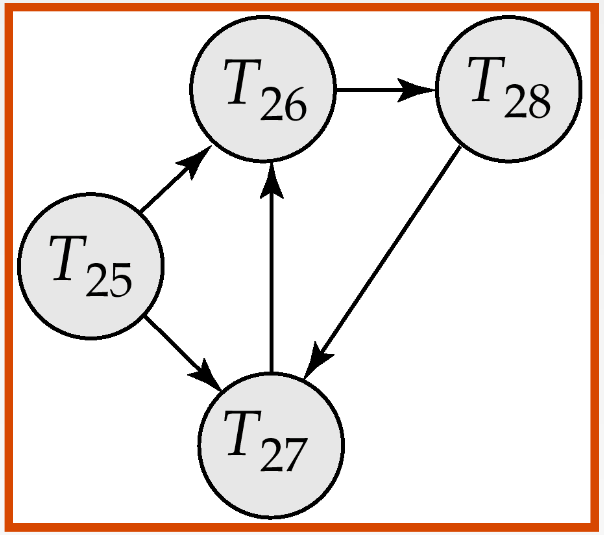
- The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.
- **Starvation** is also possible if concurrency control manager is badly designed. For example:
  - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
  - The same transaction is repeatedly rolled back due to deadlocks.
- Concurrency control manager can be designed to prevent starvation.
- Two-phase locking *does not* ensure freedom from deadlocks
  - Deadlock prevention protocols or deadlock detection mechanisms are needed!
- With detection mechanisms when deadlock is detected:
  - Some transaction will have to roll back (made a victim) to break deadlock. Select that transaction as victim that will incur minimum cost.

Deadlock Detection

- Deadlocks can be described as a *wait-for graph* where:
  - vertices are all the transactions in the system
  - There is an edge $T_i \rightarrow T_k$ in case T_i is waiting for T_k
- When T_i requests a data item currently being held by T_k, then the edge $T_i \rightarrow T_k$ is inserted in the wait-for graph. This edge is removed only when T_k is no longer holding a data item needed by T_i.
- The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles.



Wait-for graph without a cycle



Wait-for graph with a cycle

Properties of the Two-Phase Locking Protocol

- Cascading rollback is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking**. Here a transaction must hold all its exclusive locks until it commits/aborts.
- **Rigorous two-phase locking** is even stricter: here *all* locks are held until commit/abort. In this protocol transactions can be serialised in the order in which they commit.
- There can be conflict serialisable schedules that cannot be obtained if two-phase locking is used.
- However, in the absence of extra information (e.g., ordering of access to data), two-phase locking is needed for conflict serialisability in the following sense:
  - Given a transaction T_1 that does not follow two-phase locking, we can find a transaction T_2 that uses two-phase locking, and a schedule for T_1 and T_2 that is not conflict serialisable.

Timestamp-Based Protocols

- Instead of determining the order of each operation in a transaction at execution time, determines the order by the time of beginning of each transaction.
  - Each **transaction** is issued a **timestamp** when it enters the system. If an old transaction T_o has timestamp TS(T_n), a new transaction T_n is assigned time-stamp TS(T_n) such that TS(T_o) $<$TS(T_n).
  - The protocol manages concurrent execution so that the timestamps determine the serialisability order.
- In order to ensure such behaviour, the protocol maintains for each data **item** *Q* two **timestamp** values:
  - **W-timestamp**(*Q*) is the largest timestamp of any transaction that executed **write**(*Q*) successfully
    - i.e. the starting time of the transaction that wrote into Q, and started the latest
  - **R-timestamp**(*Q*) is the largest timestamp of any transaction that executed **read**(*Q*) successfully.

Timestamp-Based Protocols (Cont.)

- The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in the timestamp order.
- Suppose a transaction T issues a **read**(*Q*)
  1. If TS(*T*) < W-timestamp(*Q*), i.e. T started before the transaction that already wrote into Q, then *T* needs to read a value of *Q* that was already overwritten.
     - Hence, the **read** operation is rejected, and *T* is rolled back.
  2. If TS(*T*) ≥ W-timestamp(*Q*), then the **read** operation is executed, and **R-**timestamp(*Q*) is set to **max**(R-timestamp(*Q*), TS(*T*)).
- Suppose that transaction *T* issues **write**(*Q*)
  1. If TS(*T*) < R-timestamp(*Q*), i.e. T started before a transaction that already read the value of Q, then the value of *Q* that *T* is producing was needed previously, and the system assumed that that value would never be produced.
     - Hence, the **write** operation is rejected, and *T* is rolled back.
  2. If TS(*T*) < W-timestamp(*Q*), then *T* is attempting to write an obsolete value of *Q*.
     - Hence, this **write** operation is rejected, and *T* is rolled back.
  3. Otherwise, the **write** operation is executed, and W-timestamp(*Q*) is set to TS(*T*).

Correctness of Timestamp-Ordering Protocol

- The timestamp-ordering protocol guarantees serialisability since all the arcs in the precedence graph are of the form:

Thus, there will be no cycles in the precedence graph

- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.
- But the schedule may be non-cascade-free, and may not even be recoverable.

Multiversion Schemes

- Up to now we only considered a single copy (the most recent) of each database item.
- Multiversion schemes keep old versions of data item to increase concurrency.
  - Multiversion Timestamp Ordering
  - Multiversion Two-Phase Locking
- Basic Idea of multiversion schemes
  - Each successful **write** results in the creation of a new version of the data item written.
  - Use timestamps to label versions.
  - When a **read**(*Q*) operation is issued, select an appropriate version of *Q* based on the timestamp of the transaction, and return the value of the selected version.
  - **read**s never have to wait as an appropriate version is returned immediately.
- A drawback is that the creation of multiple versions increases storage overhead
  - Garbage collection mechanisms may be used…

Multiversion Timestamp Ordering

- Each data item Q has a sequence of versions <Q_1, Q_2,...., Q_m>. Each version Q_k contains three data fields:
  - **Content** - the value of version Q_k.
  - **W-timestamp**(Q_k) - timestamp of the transaction that created (wrote) version Q_k
  - **R-timestamp**(Q_k) - largest timestamp of the (latest) transaction that successfully read version Q_k
  - The status (active, committed,...) of the transaction that created Q_k
- When a transaction T creates a new version Q_k of Q, Q_k's W-timestamp and R-timestamp are initialised to TS(T).
- R-timestamp of Q_k is updated whenever a transaction T reads Q_k, and TS(T) > R-timestamp(Q_k).

Multiversion Timestamp Ordering (Cont)

- Suppose that transaction T issues a **read**(Q) or **write**(Q) operation. Let Q_k denote the version of Q whose write timestamp is equal to TS(T), if it exists, or the largest W-timestamp < TS(T) and the status is committed
  1. If transaction T issues a **read**(Q), then the value returned is the content of version Q_k.
  2. If transaction T issues a **write**(Q)
     1. if TS(T) < R-timestamp(Q_k), i.e. T started before the transaction that last read Q_k, then transaction T is rolled back.
     2. if TS(T) = W-timestamp(Q_k), the contents of Q_k are overwritten
     3. else a new version of Q is created.
- Observe that
  - Reads always succeed
  - A write by T is rejected if some other transaction T_2 that (in the serialisation order defined by the timestamp values) should read T's write, has already read a version created by a transaction older than T (the one that created Q_k, which has a timestamp ≤ TS(T))
- This protocol guarantees serialisability

Multiversion Two-Phase Locking

- Differentiates between read-only transactions and update transactions
- *Update transactions* acquire read and write locks, and hold all locks up to the end of the transaction. That is, update transactions follow rigorous two-phase locking.
  - Each successful **write** results in the creation of a new version of the data item written.
  - each version of a data item has a single timestamp whose value is obtained from a counter **ts-counter** that is incremented during commit processing.
- *Read-only transactions* are assigned a timestamp by reading the current value of **ts-counter** before they start execution; they follow the multiversion timestamp-ordering protocol for performing reads.

Multiversion Two-Phase Locking (Cont.)

- When an update transaction wants to read a data item:
  - it obtains a shared lock on it, and reads the latest version.
- When it wants to write an item
  - it obtains X-lock; it then creates a new version of the item and sets this version's timestamp to ∞.
    - This is to prevent other concurrent transactions to read its value, and guarantee that other reads on the same transaction get this version.
- When update transaction *T* completes, commit processing occurs:
  - *T* sets timestamp on the versions it has created to **ts-counter** + 1
  - *T* increments **ts-counter** by 1
- Read-only transactions that start after *T* incremented **ts-counter** will see the values updated by *T*.
- Read-only transactions that start before *T* incremented the **ts-counter** will see the value before the updates by *T*.
- Only serialisable schedules are produced.

Weak Levels of Consistency

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

Levels of Consistency in SQL

- **Serializable** — default in SQL standard
- **Repeatable read** — only committed records to be read, repeated reads of same record must return a same value. However, a transaction may not be serialisable – 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 a record may return different (but committed) values.
- **Read uncommitted** — even uncommitted records may be read. I.e., no isolation at all!

- In many database systems, such as Oracle, read committed is the default consistency level
  - has to be explicitly changed to serialisable when required
    - **set isolation level serializable**
- Lower degrees of consistency are useful for gathering non-critical approximate information about the database

Snapshot Isolation

- Isolation level, weaker than serialisability, that is often used by DBMSs.
  - Guarantees that all read operations in a transaction see a consistent snapshot of the database
    - Usually the snapshot has the committed values at the moment the database started (or those at the first reading operation)
  - If at the end, the write operations performed in the transaction conflict with other concurrent transaction's writes since the read snapshot, the transaction fails; otherwise succeed
- Snapshot isolation can be implemented via multi-version protocols, without locks on reads
  - This way it allows for more concurrency than serialisability
  - But may cause anomalies (*write-skews*)
- Though not in the SQL recommendation, many DBMSs adhere to it:
  - Oracle (as we shall see), SQL-Server and PostgreSQL are among those

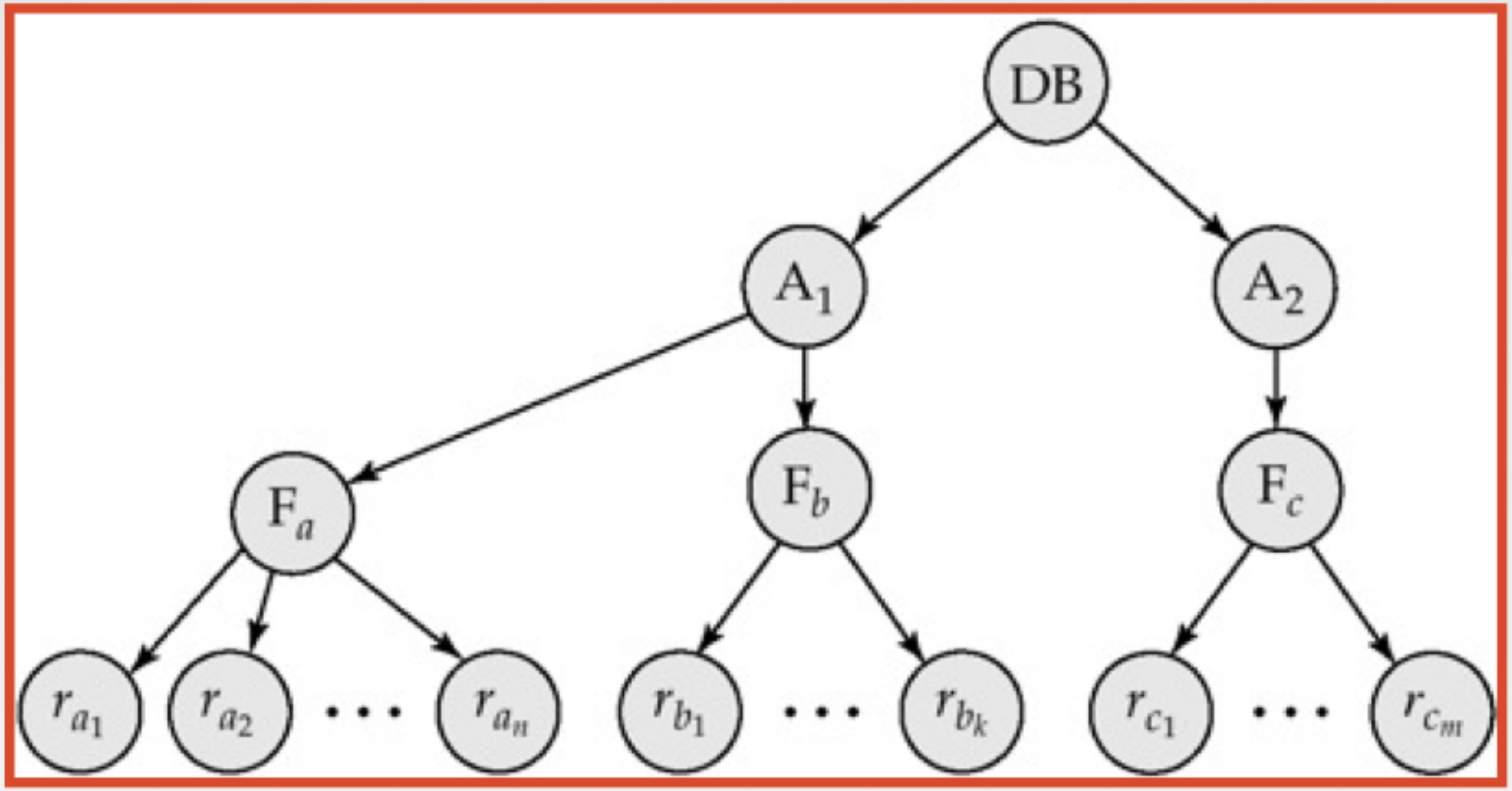
Write Skews

- Comes from failure to detect read-write conflicts
- Example:
  - Consider a database with 2 items, I1 and I2, with a constraint imposing that $I1+I2 \geq 0$.
  - At a given moment both I1 and I2 contain the number 5, and 2 concurrent transactions start
  - T1 (resp. T2) decrements I1 (resp. I2) by 10
    - Independently both transaction are consistent (in both of them, in the end I1+I2=0)
    - no write operation conflict with another write
    - So they both succeed!
  - No serialisation would succeed! (in both, in the end I1+I2 = -10)
- This can be remedied by imposing write-write conflicts
  - E.g. in the example by creating an auxiliary item storing I1+I2, that would be updated by both transactions, or also write the other item, unchanged.

Multiple Granularity

- Up to now we have considered locking (and execution) at the level of a single item/row
- However there are circumstances at which it is preferable to perform locks at different level (sets of tuples, relation, or even sets of relations)
  - As extreme example consider a transaction that needs to access to the whole database: performing locks tuple by tuple would be time-consuming
- Allow data items to be of various sizes and define a hierarchy (tree) of data granularities, where the small granularities are nested within larger ones
- When a transaction locks a node in the tree *explicitly*, it *implicitly* locks all the node's descendants in the same mode.
- Granularity of locking (level in the tree where locking is done):
  - **fine granularity** (lower in the tree): high concurrency, high locking overhead
  - **coarse granularity** (higher in the tree): low locking overhead, low concurrency

Example of Granularity Hierarchy



The levels, starting from the coarsest (top) level are

- *database*
- *area*
- *file*
- *record*

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, after previous transaction.
- 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);
- Four levels of (weak) consistency, cf. before.

Transaction management in Oracle

- Transaction beginning and ending as in SQL
  - Explicit **commit work** and **rollback work**
  - Implicit commit on session end, and implicit rollback on failure
  - Implicit commit before and after DDL commands
- Log-based deferred recovery using rollback segment
- Checkpoints (inside transactions) can be handled explicitly
  - **savepoint** <name>
  - **rollback to** <name>

- Concurrency control is made by snapshot isolation
- Deadlock are detected using a *wait-graph*
  - Upon deadlock detection, the operation locked for longer fails (but the transaction is not rolled back)

Consistency verification in Oracle

- By default, consistency is verified after each command, rather than at the end of the transaction, as is prescribed by ACID properties
- However, it is possible to defer the verification of constraints to the end of transactions
- This requires both:
  - A prior declaration of all constraints that can possibly be deferred
    - Done by adding **deferrable** to the end of the declarations of the constraint
  - an instruction in the beginning of each of the transactions where constraints are deferred
    - Done with **set constraints all deferred** or
      set constraints *<nome$_1$>, ..., <nome$_n$>* deferred

Levels of Consistency in Oracle

- Oracle implements 2 of the 4 of levels of SQL
  - *Read committed*, by default in Oracle and with
    - **set transaction isolation level read committed**
  - *Serializable* (which indeed implements *Snapshot Isolation*) with
    - **set transaction isolation level serializable**
    - Appropriate for large databases with only few updates, and usually with not many conflicts. Otherwise it is too costly.
- Further, it supports a level similar to *repeatable read*:
  - Read only mode, only allow reads on committed data, and further doesn't allow INSERT, UPDATE or DELETE on that data (without unrepeatable reads!)
    - **set transaction read only**

Granularity in Oracle

- By default Oracle performs **row level locking**.
- Command

  select … for update

  locks the selected rows so that other users cannot lock or update the rows until you end your transaction. Restriction:
  - Only at top-level select (not in sub-queries)
  - Not possible with **DISTINCT** operator, **CURSOR** expression, set operators, **group by** clause, or aggregate functions.
- Explicit locking of tables is possible in several modes, with
  - **lock table** <name> **in**
    - **row share mode**
    - **row exclusive mode**
    - **share mode**
    - **share row exclusive mode**
    - **exclusive mode**

Lock modes in Oracle

- Row share mode
  - The least restrictive mode (with highest degree of concurrency)
  - Allows other transactions to query, insert, update, delete, or lock rows concurrently in the same table, except for exclusive mode
- Row exclusive mode
  - As before, but doesn't allow setting other modes except for row share.
  - Acquired automatically after a **insert**, **update** or **delete** command on a table
- Exclusive mode
  - Only allows queries to records of the locked table
  - No modifications are allowed
  - No other transaction can lock the table in any other mode
- See manual for details of other (intermediate) modes