# Database System
## (Transactions)

**Oleh**
**Ir. I Gede Made Karma, MT**



## Transactions

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



## Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- A transaction must see a consistent database.
- During transaction execution the database may be temporarily inconsistent.
- When the transaction completes successfully (is committed), the database must be consistent.
- After a transaction commits, the changes it has made to the database persist, even if there are system failures.
- Multiple transactions can execute in parallel.
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions



## 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.



## 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, the system should ensure that its updates are not reflected in the database, else an inconsistency will result.
- **Consistency requirement** – the sum of A and B is unchanged by the execution of the transaction.



## Example of Fund Transfer (Cont.)

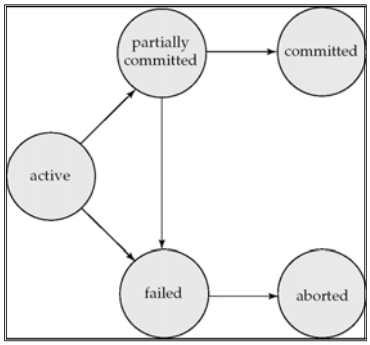
- **Isolation requirement** — if between steps 3 and 6, another transaction is allowed to access the partially updated database, it will see an inconsistent database (the sum *A* + *B* will be less than it should be).
  - Isolation can be ensured trivially by running transactions **serially**, that is one after the other.
  - However, executing multiple transactions concurrently has significant benefits, as we will see later.
- **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 despite failures.

## Transaction State

- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed --** after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - restart the transaction; can be done only if no internal logical error
  - kill the transaction
- **Committed** – after successful completion.



## Transaction State (Cont.)





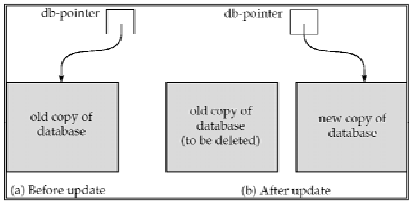

## Implementation of Atomicity and Durability

- The **recovery-management** component of a database system implements the support for atomicity and durability.
- The ***shadow-database*** scheme:
  - assume that only one transaction is active at a time.
  - a pointer called db_pointer always points to the current consistent copy of the database.
  - all updates are made on a *shadow copy* of the database, and **db_pointer** is made to point to the updated shadow copy only after the transaction reaches partial commit and all updated pages have been flushed to disk.
  - in case transaction fails, old consistent copy pointed to by **db_pointer** can be used, and the shadow copy can be deleted.



## Implementation of Atomicity and Durability (Cont.)

The shadow-database scheme:




- Assumes disks do not fail
- Useful for text editors, but
  - extremely inefficient for large databases (why?)
  - Does not handle concurrent transactions
- Will study better schemes in Chapter 17.



## Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **increased processor and disk utilization**, leading to better transaction *throughput:* one transaction can be using the CPU while another is reading from or writing to the disk
  - **reduced average response time** for transactions: short transactions need not wait behind long ones.
- **Concurrency control schemes** – mechanisms to achieve isolation; that is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database
  - Will study in Chapter 16, after studying notion of correctness of concurrent executions.



## Schedules

- **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a commit instructions as the last statement (will be omitted if it is obvious)
- A transaction that fails to successfully complete its execution will have an abort instructions as the last statement (will be omitted if it is obvious)

## 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($B$) |



## 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($B$) | |



## 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.



## 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($B$) |



## 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**
- We ignore operations other than **read** and **write** instructions, and 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 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.

| $T_1$ | $T_2$ |
|---|---|
| read($A$) | |
| write($A$) | |
| | read($A$) |
| | write($A$) |
| read($B$) | |
| write($B$) | |
| | read($B$) |
| | write($B$) |

Schedule 3

| $T_1$ | $T_2$ |
|---|---|
| read($A$) | |
| write($A$) | |
| read($B$) | |
| write($B$) | |
| | read($A$) |
| | write($A$) |
| | read($B$) |
| | write($B$) |

Schedule 6



## Conflict Serializability (Cont.)

- Example of a schedule that is not conflict serializable:

| $T_3$ | $T_4$ |
|---|---|
| read($Q$) | |
| | write($Q$) |
| write($Q$) | |

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



## 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:
  1. For each data item $Q$, if transaction $T_i$ reads the initial value of $Q$ in schedule $S$, then transaction $T_i$ must, in schedule $S'$, also read the initial value of $Q$.
  2. For each data item $Q$ if transaction $T_i$ executes **read**($Q$) in schedule $S$, and that value was produced by transaction $T_j$ (if any), then transaction $T_i$ must in schedule $S'$ also read the value of $Q$ that was produced by transaction $T_j$ .
  3. For each data item $Q$, the transaction (if any) that performs the final **write**($Q$) operation in schedule $S$ must perform the final **write**($Q$) operation in schedule $S'$.

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



## View Serializability (Cont.)

- A schedule $S$ is **view serializable** it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Below is a schedule which is view-serializable but *not* conflict serializable.

| $T_3$ | $T_4$ | $T_6$ |
|---|---|---|
| read($Q$) | | |
| | write($Q$) | |
| write($Q$) | | |
| | | write($Q$) |

- What serial schedule is above equivalent to?
- Every view serializable schedule that is not conflict serializable has **blind writes.**



## Other Notions of Serializability

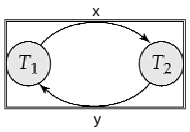
- The schedule below produces same outcome as the serial schedule < $T_1$, $T_5$ >, yet is not conflict equivalent or view equivalent to it.

| $T_1$ | $T_5$ |
|---|---|
| read($A$) | |
| $A := A - 50$ | |
| write($A$) | |
| | read($B$) |
| | $B := B - 10$ |
| | write($B$) |
| read($B$) | |
| $B := B + 50$ | |
| write($B$) | |
| | read($A$) |
| | $A := A + 10$ |
| | write($A$) |

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

## Testing for Serializability

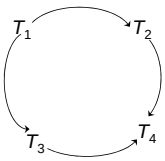
- Consider some schedule of a set of transactions $T_1$, $T_2$, ..., $T_n$
- **Precedence graph** — a direct graph where the vertices are the transactions (names).
- We draw an arc from $T_i$ to $T_j$ if the two transaction conflict, and $T_i$ accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.
- **Example 1**






## Example Schedule (Schedule A) + Precedence Graph

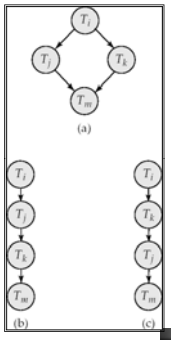
| $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_5$ |
|---|---|---|---|---|
| | read(X) | | | |
| read(Y) | | | | |
| read(Z) | | | | |
| | | | | read(V) |
| | | | | read(W) |
| | | | | read(W) |
| | read(Y) | | | |
| | write(Y) | | | |
| | | write(Z) | | |
| read(U) | | | | |
| | | | read(Y) | |
| | | | write(Y)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






## Test for Conflict Serializability

- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which take order $n^2$ time, where $n$ is the number of vertices in the graph.
  - (Better algorithms take order $n + e$ where $e$ is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.
  - This is a linear order consistent with the partial order of the graph.
  - For example, a serializability order for Schedule A would be $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$
    - Are there others?






## Test for View Serializability

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



## Recoverable Schedules

Need to address the effect of transaction failures on concurrently running transactions.

- **Recoverable schedule** — if a transaction $T_j$ reads a data item previously written by a transaction $T_i$ , then the commit operation of $T_i$ appears before the commit operation of $T_j$.
- The following schedule (Schedule 11) 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) 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 nonseralizable 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



## 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.
- Levels of consistency specified by SQL-92:
  - **Serializable** — default
  - **Repeatable read**
  - **Read committed**
  - **Read uncommitted**

## End of Chapter



| $T_1$ | $T_2$ |
|---|---|
| read($A$) | |
| write($A$) | |
| read($B$) | |
| write($B$) | |
| | read($A$) |
| | write($A$) |
| | read($B$) |
| | write($B$) |



| $T_1$ | $T_2$ |
|---|---|
| read($A$) | |
| write($A$) | |
| | read($A$) |
| read($B$) | |
| | write($A$) |
| write($B$) | |
| | read($B$) |
| | write($B$) |



## Schedule 7

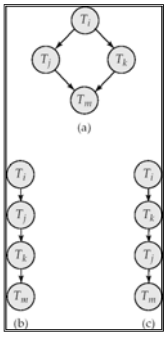
| $T_3$ | $T_4$ |
|---|---|
| read($Q$) | |
| | write($Q$) |
| write($Q$) | |



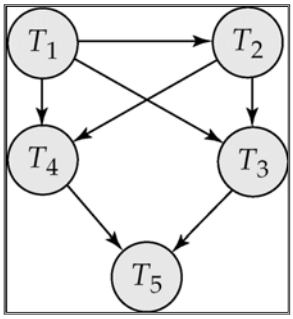
## Precedence Graph for (a) Schedule 1 and (b) Schedule 2




## Illustration of Topological Sorting



## Precedence Graph






## fig. 15.21

| $T_3$ | $T_4$ | $T_7$ |
|---|---|---|
| read($Q$) | | |
| | write($Q$) | |
| | | read($Q$) |
| write($Q$) | | |
| | | write($Q$) |



## Implementation of Isolation

- Schedules must be conflict or view serializable, and recoverable, for the sake of database consistency, and preferably cascadeless.
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency.
- Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur.
- Some schemes allow only conflict-serializable schedules to be generated, while others allow view-serializable schedules that are not conflict-serializable.



## Figure 15.6

| $T_1$ | $T_2$ |
|---|---|
| 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($B$) |



## Figure 15.12

| $T_3$ | $T_4$ | $T_6$ |
|---|---|---|
| read($Q$) | | |
| | write($Q$) | |
| write($Q$) | | |
| | | write($Q$) |