Transactions

- Transaction Concept
- ACID Properties
- Transaction State
- Concurrent Executions
- Serializability
- Recoverability
- Implementation of Isolation
- Transaction Definition in SQL
- Testing for Serializability.
A transaction is a unit of program execution that accesses and possibly updates various data items.

Less formally, a transaction typically consists of a collection of operations that someone, e.g., a programmer, wants to execute.

Up until now, we have only considered individual queries, but most non-trivial database applications perform more sophisticated, longer-running transactions.

Some implications:
- A transaction will put a database in an inconsistent state during its execution, at least temporarily.
- The probability of system failure during a transaction is non-trivial.
Additionally, DBMSs support concurrent execution of queries/transactions.

Additional implications:
- Multiple transactions may require access to the same data at the same time.
- One transaction might want to access data being modified by another.
General DBMS transaction requirements:
- A database should not get screwed up, or made inconsistent, by an individual transaction.
- Transactions should execute quickly.

More specifically:
- After a transaction completes, the database must be in a consistent state.
- After a transaction completes, the changes it has made must persist.
- Failures must not corrupt the database, even during transaction execution.
- A transaction must “see” a consistent database.
- Multiple transactions should be able to execute in parallel.
More formally, a DBMS must guarantee the ACID properties:

- **Atomicity** - All operations of a transaction are completed successfully or none are.

- **Consistency** - Transaction execution in isolation preserves database consistency.

- **Isolation** - Multiple transactions executing concurrently, must be “unaware” of each other; intermediate transaction results must be hidden from other concurrently executed transactions.
  - 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 after $T_i$ finished.

- **Durability** - After a transaction completes successfully, the changes it has made to the database persist, even in the event of system failures.
Example of Fund Transfer

Consider a transaction to transfer $50 from account A to account B:

1. begin transaction
2. read(A) // “read” means read the value from disk
3. A := A – 50 // performed in memory; could be lost by power failure
4. write(A) // “write” means write the value to disk
5. read(B)
6. B := B + 50
7. write(B)
8. end transaction;

Atomicity: If the system, and hence the transaction, fails after step 4 and before step 7, the DBMS must ensure that either the updates to A are not reflected in the database, or that the transaction gets finished when the system comes back up.
Consider a transaction to transfer $50 from account A to account B:

1. begin transaction
2. read(A) // “read” means read the value from disk
3. A := A – 50 // performed in memory
4. write(A) // “write” means write the value to disk
5. read(B)
6. B := B + 50
7. write(B)
8. end transaction;

Consistency: The sum of A and B is unchanged by the execution of the transaction, i.e., no money gets unintentionally lost.
Example of Fund Transfer

Consider a transaction to transfer $50 from account A to account B:

1. begin transaction
2. read(A)  // “read” means read the value from disk
3. A := A – 50  // performed in memory
4. write(A)  // “write” means write the value to disk
5. read(B)
6. B := B + 50
7. write(B)
8. end transaction;

Isolation: If between steps 4 and 7, another transaction is allowed to access the partially updated database, it would see an inconsistent database (the sum $A + B$ will be less than it should be).

- Isolation can be ensured trivially by running transactions **serially**.
- However, executing multiple transactions concurrently has significant benefits, as we will see later.
Example of Fund Transfer

Consider a transaction to transfer $50 from account A to account B:

1. begin transaction
2. read(A) // “read” means read the value from disk
3. A := A – 50 // performed in memory
4. write(A) // “write” means write the value to disk
5. read(B)
6. B := B + 50
7. write(B)
8. end transaction;

Durability: 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 in the even to future system failures.
A transaction goes through several states during its’ execution:

- The initial state; a transaction stays in this state while it is executing; changes are not visible to other transactions.
- The final state; final results have not been committed and are still not visible to other transactions.
- The committed state; changes are committed, i.e., they become durable and are made visible to other transactions.
- The failed state; after discovery that normal execution can no longer proceed; any changes made to the database are undone, i.e., the transaction is “rolled back.”
- The aborted state; after the database has been restored to its state prior to the start of the transaction; transaction is “killed” or restarted at this point.
So how are the ACID requirements implemented?

Atomicity and durability are implemented by the *recovery-management* subsystem.

A simplistic approach to recovery-management is the *shadow-database* scheme:

- 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 (active and partially committed).
- *db_pointer* is updated only after all updates have been written to disk (commit).
- If the transaction fails, the old copy pointed to by *db_pointer* is retained, and the shadow copy is deleted.

![Diagram of shadow-database scheme](image)
Notes:

- Extremely inefficient for large databases.
- Assumes that at most one modifying transaction is executing at a time.
- Assumes that writing \textit{db_pointer} is an atomic operation (probably true).
- Used for some text, photo and video editors, single-user databases, and other simple programs.

More sophisticated schemes will be discussed in the next chapter.
Concurrent Executions

How about isolation?

The easiest way to guarantee isolation is to execute transactions serially.

From a performance perspective, this is too restrictive. Multiple transactions should be allowed to run concurrently.

- Increased throughput
- Reduced average response time
Given a set of transactions, where each transaction consists of a sequence of instructions, a schedule is a sequence of instructions for the transactions specified in chronological order of execution.

Suppose $T_1$ transfers $50$ from $A$ to $B$, and $T_2$ transfers $10\%$ from $A$ to $B$.

The following is a serial schedule (#1) where $T_1$ is followed by $T_2$:

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($A$)</td>
<td>read($A$)</td>
</tr>
<tr>
<td>$A := A - 50$</td>
<td>temp := $A * 0.1$</td>
</tr>
<tr>
<td>write($A$)</td>
<td>$A := A - \text{temp}$</td>
</tr>
<tr>
<td>read($B$)</td>
<td>write($A$)</td>
</tr>
<tr>
<td>$B := B + 50$</td>
<td>read($B$)</td>
</tr>
<tr>
<td>write($B$)</td>
<td>write($B$)</td>
</tr>
</tbody>
</table>

Time
The following is also a serial schedule (#2) where $T_2$ is followed by $T_1$:

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($A$)</td>
<td>read($A$)</td>
</tr>
<tr>
<td>$A := A - 50$</td>
<td>temp := $A \times 0.1$</td>
</tr>
<tr>
<td>write($A$)</td>
<td>$A := A - \text{temp}$</td>
</tr>
<tr>
<td>read($B$)</td>
<td>write($A$)</td>
</tr>
<tr>
<td>$B := B + \text{temp}$</td>
<td>read($B$)</td>
</tr>
<tr>
<td>write($B$)</td>
<td>$B := B + \text{temp}$</td>
</tr>
</tbody>
</table>

Note that the result of the two schedules is generally different.  

- So which is “correct”?
If a particular order is required for two transactions, then that implies they should be one transaction that reflects the required order.

In other words, either of the previous serial schedules is “correct.”

They might give different results, but that’s ok…that’s what happens in the real world; people and processes don’t always coordinate.
So what does a *concurrent* schedule look like?

Let $T_1$ and $T_2$ be the transactions defined previously.

The following concurrent schedule is not a serial schedule, but is *equivalent* to Schedule 1.

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($A$)</td>
<td>read($A$)</td>
</tr>
<tr>
<td>$A := A - 50$</td>
<td>$temp := A * 0.1$</td>
</tr>
<tr>
<td>write($A$)</td>
<td>$A := A - temp$</td>
</tr>
<tr>
<td>read($B$)</td>
<td>read($B$)</td>
</tr>
<tr>
<td>$B := B + 50$</td>
<td>$B := B + temp$</td>
</tr>
<tr>
<td>write($B$)</td>
<td>write($B$)</td>
</tr>
</tbody>
</table>
The following is also a concurrent schedule, but it is not equivalent to either schedule #1 or #2:

Consider what happens when A and B both initially contain $100.

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>read(A)</td>
</tr>
<tr>
<td>A := A - 50</td>
<td>temp := A * 0.1</td>
</tr>
<tr>
<td>write(A)</td>
<td>A := A - temp</td>
</tr>
<tr>
<td>read(B)</td>
<td>write(A)</td>
</tr>
<tr>
<td></td>
<td>read(B)</td>
</tr>
<tr>
<td>B := B + 50</td>
<td></td>
</tr>
<tr>
<td>write(B)</td>
<td>B := B + temp</td>
</tr>
<tr>
<td></td>
<td>write(B)</td>
</tr>
</tbody>
</table>
Equivalence to a serial schedule is imperative! (why?)

Which one doesn’t matter!
So, in addition to concurrency, what do we require of our schedules?

A schedule for a set of transactions must:
- Include all transaction instructions.
- Preserve the order in which the instructions appear in each individual transaction

Additionally, it is assumed that each transaction preserves database consistency.

Thus serial execution of a set of transactions preserves database consistency.
A (concurrent) schedule is said to be *serializable* if it is “equivalent” to a serial schedule.

There are two different forms of schedule equivalence:

1. conflict serializability
2. view serializability

For the sake of simplicity, operations other than reads and writes are ignored.

- It is assumed (conservatively) that transactions may perform arbitrary computations between reads and writes.
- This will result in false negatives, but not false positives…huh?
Conflicting Instructions

Instructions $l_i$ and $l_j$ of transactions $T_i$ and $T_j$, respectively, conflict if there exists an item $Q$ accessed by both $l_i$ and $l_j$, and at least one of these instructions wrote $Q$.

1. $l_i = \text{read}(Q)$, $l_j = \text{read}(Q)$ - $l_i$ and $l_j$ don’t conflict
2. $l_i = \text{read}(Q)$, $l_j = \text{write}(Q)$ - conflict
3. $l_i = \text{write}(Q)$, $l_j = \text{read}(Q)$ - conflict
4. $l_i = \text{write}(Q)$, $l_j = \text{write}(Q)$ - conflict

Intuitively, a conflict between $l_i$ and $l_j$ means that their relative order of execution can make a difference.

On the other hand, if $l_i$ and $l_j$ do not conflict, then their relative order of execution cannot make a difference.
Let $S$ and $S'$ be two schedules.

If $S$ can be transformed into $S'$ by a series of “swaps” of non-conflicting instructions, then $S$ and $S'$ are conflict equivalent.

Equivalently, $S$ and $S'$ are said to be conflict equivalent if the order of all pairs of conflicting instructions are the same in the two schedules.

We say that a schedule $S$ is conflict serializable if it is conflict equivalent to a serial schedule.
Schedule A can be transformed into Schedule B (a serial schedule) where $T_2$ follows $T_1$, by series of swaps of non-conflicting instructions.

Therefore Schedule A is conflict serializable.

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($A$)</td>
<td>read($A$)</td>
</tr>
<tr>
<td>write($A$)</td>
<td>write($A$)</td>
</tr>
<tr>
<td>read($B$)</td>
<td>read($B$)</td>
</tr>
<tr>
<td>write($B$)</td>
<td>write($B$)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($A$)</td>
<td>read($A$)</td>
</tr>
<tr>
<td>write($A$)</td>
<td>write($A$)</td>
</tr>
<tr>
<td>read($B$)</td>
<td>read($B$)</td>
</tr>
<tr>
<td>write($B$)</td>
<td>write($B$)</td>
</tr>
</tbody>
</table>
Example of a schedule that is not conflict serializable:

<table>
<thead>
<tr>
<th>$T_3$</th>
<th>$T_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($Q$)</td>
<td>write($Q$)</td>
</tr>
<tr>
<td>write($Q$)</td>
<td></td>
</tr>
</tbody>
</table>

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 Conflict Serializability

Consider a schedule for a set of transactions $T_1, T_2, ..., T_n$

A **precedence graph** for the schedule is a directed graph which has:

- A vertex for each transaction
- An edge from $T_i$ to $T_j$ if they contain conflicting instructions, and the conflicting instruction from $T_i$ accessed the data item before the conflicting instruction from $T_j$
  - If executes $T_i$ write($Q$) before $T_j$ executes read($Q$)
  - If executes $T_i$ read($Q$) before $T_j$ executes write($Q$)
  - If executes $T_i$ write($Q$) before $T_j$ executes write($Q$).

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

Duplicate edges may result from the above, but can be deleted.
Example:

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($A$)</td>
<td>read($A$)</td>
</tr>
<tr>
<td>$A := A - 50$</td>
<td>temp := $A * 0.1$</td>
</tr>
<tr>
<td>write($A$)</td>
<td>$A := A - \text{temp}$</td>
</tr>
<tr>
<td>read($B$)</td>
<td>write($A$)</td>
</tr>
<tr>
<td>$B := B + 50$</td>
<td>read($B$)</td>
</tr>
<tr>
<td>write($B$)</td>
<td>$B := B + \text{temp}$</td>
</tr>
</tbody>
</table>

Graph:

- $T_1$ to $T_2$
Example Schedule (Schedule A) and Precedence Graph
Test for Conflict Serializability

- **Observation:** A schedule is conflict serializable 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 $O(n + e)$ where $e$ is the # of edges.

- If a precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* 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?

- Special case – what if each transaction operates (reads/writes) on a unique variable?
The preceding definition of a serializable schedule is very conservative in that false negatives will be detected.

In other words, schedules will be deemed non-serializable that in fact are.

This leads to a more relaxed definition of serializability…
Let \( S \) and \( S´ \) be two schedules with the same set of transactions.

\( S \) and \( S´ \) are said to be \textit{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 \( T_i \) reads the initial value of \( Q \) in schedule \( S´ \).
2. For each data item \( Q \), the transaction (if any) that performs the final \texttt{write}(\( Q \)) operation in schedule \( S \) performs the final \texttt{write}(\( Q \)) operation in schedule \( S´ \).
3. For each data item \( Q \), if transaction \( T_i \) executes \texttt{read}(\( Q \)) in schedule \( S \), and that value was produced by transaction \( T_j \) in \( S \), then transaction \( T_i \) reads the value of \( Q \) that was produced by transaction \( T_j \) in schedule \( S´ \).

View equivalence is also based only on reads and writes, but it considers details of those reads and writes a bit more closely:

- It is therefore less conservative, i.e., fewer false negatives.
A schedule $S$ is **view-serializable** if it is view-equivalent to a serial schedule.

Every conflict-serializable schedule is also view-serializable (prove as an exercise).

- All previous conflict-serializable schedules are therefore view-serializable.

However, a schedule may be view-serializable, but not conflict-serializable.
Example #1:

<table>
<thead>
<tr>
<th></th>
<th>$T_3$</th>
<th>$T_4$</th>
<th>$T_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($Q$)</td>
<td>write($Q$)</td>
<td>write($Q$)</td>
<td></td>
</tr>
<tr>
<td>write($Q$)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- What serial schedule is the above view-equivalent to?
- Is the above schedule conflict-equivalent to that serial schedule?
- What is the point?
Example #2:

<table>
<thead>
<tr>
<th>$T_3$</th>
<th>$T_4$</th>
<th>$T_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($Q$)</td>
<td>write($Q$)</td>
<td>read($Q$)</td>
</tr>
<tr>
<td>write($Q$)</td>
<td></td>
<td>write($Q$)</td>
</tr>
</tbody>
</table>

What serial schedule is the above view-equivalent to?

Is the above schedule conflict-equivalent to that serial schedule?

Every view-serializable schedule that is not conflict-serializable must contain at least one *blind write* (a write of a variable without first reading that variable; prove as an exercise).
As described, the precedence graph test for conflict serializability cannot be used to test for view serializability.

The test can be modified to work for view serializability, but has cost exponential in the size of the precedence graph (see the example on the class website).

In fact, the problem of checking if a schedule is view-serializable is \( NP \)-complete.

- Thus existence of an efficient algorithm is extremely unlikely.
Regardless, even view-serializability is too conservative, however.

The schedule below produces the same outcome as the serial schedule \( T_1, T_5 \) yet is not conflict equivalent or view equivalent to it.

\[
\begin{array}{c|c}
T_1 & T_5 \\
\hline
\text{read}(A) & \text{read}(B) \\
A := A - 50 & B := B - 10 \\
\text{write}(A) & \text{write}(B) \\
\hline
\text{read}(B) & \text{read}(A) \\
B := B + 50 & A := A + 10 \\
\text{write}(B) & \text{write}(A) \\
\end{array}
\]

Similarly for the schedule \( < T_5, T_1 > \)

Determining such equivalence requires analysis of operations other than reads and writes; this is why we said earlier that our approach was “conservative.”
So far we have assumed that transactions execute to completion, i.e., there are no transaction failures.

What is the effect of transaction failures on concurrently running transactions?

Consider the following schedule, which is conflict and, hence, view serializable.

<table>
<thead>
<tr>
<th></th>
<th>$T_8$</th>
<th>$T_9$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>read(A)</td>
<td>read(A)</td>
</tr>
<tr>
<td></td>
<td>write(A)</td>
<td>write(C)</td>
</tr>
<tr>
<td></td>
<td>read(B)</td>
<td>write(C)</td>
</tr>
</tbody>
</table>
Now suppose that $T_9$ commits immediately after the write, but before $T_8$ commits.

<table>
<thead>
<tr>
<th>$T_8$</th>
<th>$T_9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>read(A)</td>
</tr>
<tr>
<td>write(A)</td>
<td>write(C)</td>
</tr>
<tr>
<td>read(B)</td>
<td>commit</td>
</tr>
<tr>
<td>commit</td>
<td></td>
</tr>
</tbody>
</table>
Now suppose $T_8$ aborts instead of committing.

In that case $T_9$ would read (and possibly write or display) an inconsistent database state.

<table>
<thead>
<tr>
<th>$T_8$</th>
<th>$T_9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A) write(A)</td>
<td>read(A) write(C) commit</td>
</tr>
<tr>
<td>read(B) abort</td>
<td></td>
</tr>
</tbody>
</table>

The above schedule is said to be *non-recoverable*. 
A schedule is said to be *recoverable* if whenever a transaction $T_j$ reads a data item previously written by a transaction $T_i$, the `commit` operation of $T_i$ appears before the `commit` operation of $T_j$.

Given a schedule, how can we determine if the schedule is recoverable?

- Label each vertex in the precedence graph with the time of the commit.
- If there is an edge from $v_i$ to $v_j$, then the time on $v_j$ must be after the time on $v_i$.

The DBMS must ensure that all concurrent schedules are recoverable by imposing the above ordering on transaction commits.

However, even if the DBMS does ensure that schedules are recoverable, significant rework might still occur in the context of an abort.
A *cascading rollback* is 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).

<table>
<thead>
<tr>
<th></th>
<th>$T_{10}$</th>
<th>$T_{11}$</th>
<th>$T_{12}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>read($A$)</td>
<td>read($A$)</td>
<td>read($A$)</td>
</tr>
<tr>
<td>Read</td>
<td>read($B$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write</td>
<td>write($A$)</td>
<td>write($A$)</td>
<td></td>
</tr>
</tbody>
</table>

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

This can lead to the undoing of a significant amount of work.
A schedule is said to be *cascadeless* if 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 $T_j$.

Cascading rollbacks cannot occur in a cascadeless schedule.

Every cascadeless schedule is also recoverable (why?).
Focusing only on reads and writes simplifies the process, but may lead to false negatives, thereby hindering concurrency.

- Conflict-serializability leads to false negatives, thereby reducing concurrency.
- Similarly for view-serializability but not as bad as for conflict-serializability.

Requiring recoverable schedules hinders concurrency more…

Requiring cascade-less schedules hinders concurrency even more…

So why are we doing this? Aren’t we trying to encourage concurrency?

We are trying to restrict concurrent schedules in order to guarantee isolation and recoverability, while at the same time maximizing concurrency.
Weak Levels of Consistency

- Some applications are willing to live with weaker levels of isolation, allowing schedules that are not serializable:
  - A read-only transaction that calculates an approximate total account balance.
  - Database statistics for query optimization can be approximate.

- Most DBMSs allow the user to select the level of isolation.

- A given DMBS, however, may choose to execute a query at a higher level of isolation.
Levels of Consistency in SQL

- A dirty read occurs when a transaction reads a value from a row that has been modified by another transaction but has not yet been committed.

- A non-repeatable read occurs when a transaction reads the same record more than once, and gets different (possibly committed) values.

- A phantom read occurs when a transaction issues the same select query twice, and gets a different set of (possibly committed) rows each time.
Levels of Consistency in SQL

- **Conflict serializable:**
  - No dirty reads.
  - No non-repeatable reads.
  - No phantom reads.

- **Repeatable read:**
  - No dirty reads.
  - No non-repeatable reads.
  - Allows phantom reads.
Levels of Consistency in SQL

- **Read committed:**
  - No dirty reads.
  - Allows non-repeatable reads.
  - Allows phantom reads.

- **Read uncommitted:**
  - Allows dirty reads.
  - Allows non-repeatable reads.
  - Allows phantom reads.
Concurrency Control

- A DBMS must provide a mechanism that will ensure that all schedules are:
  - Serializable (at some level)
  - Recoverable, and preferably cascadeless.

- Testing a schedule for serializability after it has executed is a little too late!

- Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless.
  - Without building or examining the precedence graph.

- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.
By the way…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?
End of Chapter