Chapter 14: Transactions
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
ACID

• Transactions must obey:
  – Atomicity
  – Consistency
  – Isolation
  – Durability

• Key acronym to remember for exams/jobs

• Details...soon
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
    • 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. \text{read}(A)
  2. \text{A} := \text{A} - 50
  3. \text{write}(A)
  4. \text{read}(B)
  5. \text{B} := \text{B} + 50
  6. \text{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
  - 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.

- During transaction execution the database may be temporarily inconsistent.
- When the transaction completes successfully the database must be consistent
Example of Fund Transfer (Cont.)

- **Isolation requirement** — if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).

  T1                      T2
  1. read($A$)       1. read($A$), read($B$), print($A+B$)
  2. $A := A - 50$
  3. write($A$)       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

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

• Multiple transactions are allowed to run concurrently in the system. Advantages are:
  – *increased processor and disk utilization*, 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
Schedules

- **Schedule** – a sequence 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 $T_2$:

<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>write ($A$)</td>
</tr>
<tr>
<td>read ($B$)</td>
<td>$A := A - temp$</td>
</tr>
<tr>
<td>$B := B + 50$</td>
<td>read ($B$)</td>
</tr>
<tr>
<td>write ($B$)</td>
<td>$B := B + temp$</td>
</tr>
<tr>
<td>commit</td>
<td>commit</td>
</tr>
</tbody>
</table>
• A serial schedule 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>$A := A - 0.1$</td>
</tr>
<tr>
<td>write $(A)$</td>
<td>write $(A)$</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>
<tr>
<td>commit</td>
<td>commit</td>
</tr>
</tbody>
</table>
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.

<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></td>
<td>write (A)</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>
<tr>
<td>commit</td>
<td>commit</td>
</tr>
</tbody>
</table>

In Schedules 1, 2 and 3, the sum \( A + B \) is preserved.
Schedule 4

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

<table>
<thead>
<tr>
<th></th>
<th>(T_1)</th>
<th>(T_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>read ((A))</td>
<td>read ((A))</td>
</tr>
<tr>
<td></td>
<td>(A := A - 50)</td>
<td>(temp := A * 0.1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(A := A - temp)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>write ((A))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>read ((B))</td>
</tr>
<tr>
<td></td>
<td>write ((A))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>read ((B))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(B := B + 50)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>write ((B))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>commit</td>
<td>commit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(B := B + temp)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>write ((B))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>commit</td>
</tr>
</tbody>
</table>
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.
Serializability

• A schedule is called *serializable* if its final effect is the same as that of a *serial schedule*

• Serializability $\rightarrow$ schedule is fine and does not result in inconsistent database
  – Since serial schedules are fine

• Non-serializable schedules are unlikely to result in consistent databases

• We will ensure serializability
  – Typically relaxed in real high-throughput environments
Serializability

- Not possible to look at all n! serial schedules to check if the effect is the same
  - Instead we ensure serializability by allowing or not allowing certain schedules

- Conflict serializability

- View serializability

- View serializability allows more schedules
Conflicting Instructions

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

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

- Intuitively, a conflict between $l_i$ and $l_j$ forces a (logical) temporal order between them.
  - If $l_i$ and $l_j$ are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.
Conflict Serializability

- If a schedule $S$ can be transformed into a schedule $S'$ by a series of swaps of non-conflicting instructions, we say that $S$ and $S'$ are *conflict equivalent*.

- 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.

<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>

Schedule 3

<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>

Schedule 6
Example of a schedule that is not conflict serializable:

<table>
<thead>
<tr>
<th></th>
<th>( T_3 )</th>
<th>( T_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>read ( (Q) )</td>
<td></td>
<td>write ( (Q) )</td>
</tr>
<tr>
<td>write ( (Q) )</td>
<td></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 \( < T_4, T_3 > \).
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 $S'$ transaction $T_i$ must also read the initial value of $Q$.

2. 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 $S'$ transaction $T_i$ must also read the value of $Q$ that was produced by the same $\text{write}(Q)$ operation of transaction $T_j$.

3. The transaction (if any) that performs the final $\text{write}(Q)$ operation in schedule $S$ must also perform the final $\text{write}(Q)$ operation in schedule $S'$.

As can be seen, view equivalence is also based purely on **reads** and **writes** alone.
• A schedule $S$ is **view serializable** if 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.

<table>
<thead>
<tr>
<th></th>
<th>$T_{27}$</th>
<th>$T_{28}$</th>
<th>$T_{29}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read $(Q)$</td>
<td></td>
<td>write $(Q)$</td>
<td></td>
</tr>
<tr>
<td>write $(Q)$</td>
<td></td>
<td></td>
<td>write $(Q)$</td>
</tr>
</tbody>
</table>

• 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 the same outcome as the serial schedule \( < T_1, T_5 > \), yet is not conflict equivalent or view equivalent to it.

<table>
<thead>
<tr>
<th>( T_1 )</th>
<th>( T_5 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>read ((A))</td>
<td>read ((B))</td>
</tr>
<tr>
<td>( A := A - 50 )</td>
<td>( B := B - 10 )</td>
</tr>
<tr>
<td>write ((A))</td>
<td>write ((B))</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>read ((B))</td>
<td>read ((A))</td>
</tr>
<tr>
<td>( B := B + 50 )</td>
<td>( A := A + 10 )</td>
</tr>
<tr>
<td>write ((B))</td>
<td>write ((A))</td>
</tr>
</tbody>
</table>

- 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**

![Diagram showing a precedence graph with nodes $T_1$ and $T_2$ and an arc from $T_1$ to $T_2$.]
Precedence graph

- Edge Ti -> Tj exists if one of the following holds:
  - Ti executes `write(Q)` before Tj executes `read(Q)`
  - Ti executes `read(Q)` before Tj executes `write(Q)`
  - Ti executes `write(Q)` before Tj executes `write(Q)`
Example Schedule (Schedule A) + Precedence Graph

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
<th>$T_4$</th>
<th>$T_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(Y)</td>
<td>read(X)</td>
<td>read(Y)</td>
<td>read(V)</td>
<td></td>
</tr>
<tr>
<td>read(Z)</td>
<td></td>
<td>write(Y)</td>
<td>read(W)</td>
<td>write(U)</td>
</tr>
<tr>
<td></td>
<td>read(U)</td>
<td>write(Z)</td>
<td>read(Y)</td>
<td>write(Z)</td>
</tr>
<tr>
<td></td>
<td>read(U)</td>
<td></td>
<td></td>
<td>read(U)</td>
</tr>
<tr>
<td></td>
<td>write(U)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Precedence Graph]

- $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_5$
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 $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$.
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.
Recoverability

- Serializability is good for consistency

- But what if transactions fail?
  - T2 has already committed
    - A user might have been notified
  - Now T1 abort creates a problem
    - T2 has seen its effect, so just aborting T1 is not enough. T2 must be aborted as well (and possibly restarted)
    - But T2 is committed

<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>tmp = A*0.1</td>
</tr>
<tr>
<td>write(A)</td>
<td>A = A – tmp</td>
</tr>
<tr>
<td></td>
<td>write(A)</td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
</tr>
</tbody>
</table>
Recoverability

- Recoverable schedule: If $T_1$ has read something $T_2$ has written, $T_2$ must commit before $T_1$
  - Otherwise, if $T_1$ commits, and $T_2$ aborts, we have a problem

- Cascading rollbacks: If $T_{10}$ aborts, $T_{11}$ must abort, and hence $T_{12}$ must abort and so on.

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

- **Dirty read**: Reading a value written by a transaction that hasn’t committed yet

- Cascadeless schedules:
  - A transaction only reads *committed* values.
  - So if T1 has written A, but not committed it, T2 can’t read it.
    - *No dirty reads*

- Cascadeless $\rightarrow$ No cascading rollbacks
  - That’s good
  - We will try to guarantee that as well
Recap

• We discussed:
  – Serial schedules, serializability
  – Conflict-serializability, view-serializability
  – How to check for conflict-serializability
  – Recoverability, cascade-less schedules

• We haven’t discussed:
  – How to guarantee serializability?
    • Allowing transactions to run, and then aborting them if the schedules wasn’t serializable is clearly not the way to go
  – We instead use schemes to guarantee that the schedule will be conflict-serializable
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 (Cont.)

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