Goal: Integrity Across Sequence of Operations

- Update should complete entirely
  - update stipend set stipend = stipend*1.03;
  - What if it gets halfway and the machine crashes?
- What about multiple operations?
  - Withdraw x from Account1
  - Deposit x into Account2
- Simultaneous operations?
  - Print paychecks while stipend being updated
Solution: Transaction

• Sequence of operations grouped into a transaction
  – Externally viewed as Atomic: All happens at once
  – DBMS manages so even the programmer gets this view

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 properties

Transactions have:

• Atomicity
  – All or nothing
• Consistency
  – Changes to values maintain integrity
• Isolation
  – Transaction occurs as if nothing else happening
• Durability
  – Once completed, changes are permanent

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.)

- **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
    - Erroneous transaction logic can lead to inconsistency

Example of Fund Transfer (Cont.)

- **Isolation requirement** — if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).
  - T1
    1. read(A)
    2. $A := A - 50$
    3. write(A)
  - T2
    4. read(A), read(B), print(A+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

A transaction is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
  - That is, for every pair of transactions $T_i$ and $T_j$, it appears to $T_i$ that either $T_j$ finished execution before $T_i$ started, or $T_j$ started execution after $T_i$ finished.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Transaction State

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

- Partially committed
- Committed
- Active
- Failed
- Aborted

Chapters 16-17
Concurrency Control

T1, T2, ..., Tn

DB (consistency constraints)
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
    - Will study in Chapter 15, after studying notion of correctness of concurrent executions.

Example:

<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+100</td>
<td>A ← A×2</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+100</td>
<td>B ← B×2</td>
</tr>
<tr>
<td>Write(B)</td>
<td>Write(B)</td>
</tr>
</tbody>
</table>

Constraint: A=B
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 instruction as the last statement
  - By default, transactions 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 A

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A); A ← A+100</td>
<td>Read(A);A ← A×2;</td>
</tr>
<tr>
<td>Write(A);</td>
<td>Write(A);</td>
</tr>
<tr>
<td>Read(B); B ← B+100;</td>
<td>Read(B);B ← B×2;</td>
</tr>
<tr>
<td>Write(B);</td>
<td>Write(B);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A);</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read(B);</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read(A);</td>
<td></td>
<td>125</td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read(B);</td>
<td></td>
<td>250</td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

11/2/2021
### Schedule B

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>B</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td><strong>Read(A); A ← A\times 2; Write(A);</strong></td>
<td><strong>Read(A); A ← A\times 2; Write(A);</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Read(B); B ← B\times 2; Write(B);</strong></td>
<td><strong>Read(B); B ← B\times 2; Write(B);</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Read(A); A ← A+100; Write(A);</strong></td>
<td><strong>Read(A); A ← A+100; Write(A);</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Read(B); B ← B+100; Write(B);</strong></td>
<td><strong>Read(B); B ← B+100; Write(B);</strong></td>
</tr>
</tbody>
</table>

11/2/2021

### Schedule C

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>B</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td><strong>Read(A); A ← A+100; Write(A);</strong></td>
<td><strong>Read(A); A ← A\times 2; Write(A);</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Read(B); B ← B+100; Write(B);</strong></td>
<td><strong>Read(B); B ← B\times 2; Write(B);</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Read(A); A ← A+100; Write(A);</strong></td>
<td><strong>Read(A); A ← A+100; Write(A);</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Read(B); B ← B+100; Write(B);</strong></td>
<td><strong>Read(B); B ← B+100; Write(B);</strong></td>
</tr>
</tbody>
</table>

11/2/2021
**Schedule D**

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A); A ← A+100</td>
<td></td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read(A); A ← A×2;</td>
<td></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td>250</td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B×2;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B+100;</td>
<td></td>
<td>250</td>
<td></td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
<td>150</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>250</td>
<td>150</td>
</tr>
</tbody>
</table>

11/2/2021

**Schedule E**

Same as Schedule D
but with new T2’

<table>
<thead>
<tr>
<th>T1</th>
<th>T2’</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A); A ← A+100</td>
<td></td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read(A); A ← A×1;</td>
<td></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B×1;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B+100;</td>
<td></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
<td>125</td>
<td></td>
</tr>
</tbody>
</table>

11/2/2021

© 2021 Christopher W. Clifton
Schedules and Concurrency

- Want schedules that are “good”, regardless of
  - initial state and
  - transaction semantics
- Only look at order of read and writes

- Example:
  - Sc=r1(A)w1(A)r2(A)w2(A)r1(B)w1(B)r2(B)w2(B)

Example

Sc=r1(A)w1(A)r2(A)w2(A)r1(B)w1(B)r2(B)w2(B)

Sc’=r1(A)w1(A)\ r_1(B)w_1(B)r_2(A)w_2(A)r_2(B)w_2(B)

T_1 \quad T_2
However, for Sd:
\[ Sd = r_1(A)w_1(A)r_2(A)w_2(A) r_2(B)w_2(B)r_1(B)w_1(B) \]

- as a matter of fact,
  \( T_2 \) must precede \( T_1 \)
in any equivalent schedule,
i.e., \( T_2 \rightarrow T_1 \)

- \( T_2 \rightarrow T_1 \)
- Also, \( T_1 \rightarrow T_2 \)

\( Sd \) cannot be rearranged
into a serial schedule
\( Sd \) is not “equivalent” to
any serial schedule
\( Sd \) is “bad”
Returning to Sc

\[ Sc=r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B) \]

\[ T_1 \rightarrow T_2 \quad T_1 \rightarrow T_2 \]

- no cycles \( \Rightarrow \) Sc is “equivalent” to a serial schedule
  (in this case \( T_1, T_2 \))

Concepts

**Transaction**: sequence of \( r_i(x), w_i(x) \) actions

**Conflicting actions**: \( r_1(A) \quad w_2(A) \quad w_1(A) \quad w_2(A) \quad r_1(A) \quad w_2(A) \)

**Schedule**: represents chronological order in which actions are executed

**Serial schedule**: no interleaving of actions or transactions
What about concurrent actions?

- So net effect is either
  - $S=\ldots r_1(x)\ldots w_2(b)\ldots$ or
  - $S=\ldots w_2(B)\ldots r_1(x)\ldots$
What about conflicting, concurrent actions on same object?

- Assume equivalent to either $r_1(A) w_2(A)$
  or $w_2(A) r_1(A)$
- $\Rightarrow$ low level synchronization mechanism
- Assumption called “atomic actions”

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**
- Simplifying assumptions
  - 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 $l_i$ 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.

Definition

- S1, S2 are conflict equivalent schedules
  - if S1 can be transformed into S2 by a series of swaps on non-conflicting actions.
- A schedule is conflict serializable if it is conflict equivalent to some 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

Conflict Serializability (Cont.)

- 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 $<T_4, T_3>$. 
View Serializability

- Let $S$ and $S'$ be two schedules with the same set of transactions. $S$ and $S'$ are view equivalent if the following three conditions are met, for each data item $Q$,
  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 $S'$.

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

View Serializability (Cont.)

- 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>$T_{37}$</th>
<th>$T_{28}$</th>
<th>$T_{29}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read ($Q$)</td>
<td>write ($Q$)</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) (A := A - 50) write (A)</td>
<td>read (B) (B := B - 10) write (B)</td>
</tr>
<tr>
<td>read (B) (B := B + 50) write (B)</td>
<td>read (A) (A := A + 10) 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 \(T_1, T_2, \ldots, T_n\)

- **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 \(T_j\) accessed the data item on which the conflict arose earlier.

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

- Example of a precedence graph
Exercise:

- What is $P(S)$ for
  $S = w_3(A) \; w_2(C) \; r_1(A) \; w_1(B) \; r_1(C) \; w_2(A) \; r_4(A) \; w_4(D)$

- Is $S$ serializable?

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?
Precedence Graphs and Conflict Equivalence: Lemma

- S1, S2 conflict equivalent \( \Rightarrow P(S1)=P(S2) \)

Proof sketch:
Assume \( P(S_1) \neq P(S_2) \)
\[ \Rightarrow \exists T_i: T_i \rightarrow T_j \text{ in } S_1 \text{ and not in } S_2 \]
\[ \Rightarrow S_1 = \ldots p_i(A)\ldots q_j(A)\ldots \]
\[ S_2 = \ldots q_j(A)\ldots p_i(A)\ldots \]
\[ \text{conflict} \]

\[ \Rightarrow S_1, S_2 \text{ not conflict equivalent} \]

Note: \( P(S1)=P(S2) \not\Rightarrow S1, S2 \text{ conflict equivalent} \)

Counter example:

\[ S_1 = w_1(A) \ r_2(A) \ w_2(B) \ r_1(B) \]

\[ S_2 = r_2(A) \ w_1(A) \ r_1(B) \ w_2(B) \]
Theorem

• P(S1) acyclic ⇐⇒ S1 conflict serializable

(⇐) Assume S1 is conflict serializable
⇒ ∃ S_s: S_s, S1 conflict equivalent
⇒ P(S_s) = P(S1)
⇒ P(S1) acyclic since P(S_s) is acyclic

Theorem:
P(S1) acyclic ⇐⇒ S1 conflict serializable

(⇒) Assume P(S1) is acyclic
Transform S1 as follows:
(1) Take T1 to be transaction with no incident arcs
(2) Move all T1 actions to the front
    S1 = .......... qj(A)........p1(A).....
(3) we now have S1 = < T1 actions ><... rest ...>
(4) repeat above steps to serialize rest!
**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.

**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
  - What do we do if the schedule *isn’t* serializable?
- Testing a schedule for serializability *after* it has executed is a little too late!
- **Goal** – to develop concurrency control protocols that will assure serializability.
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

  | \( T_8 \) | \( T_9 \) |
  | read \((A)\) | read \((A)\) |
  | write \((A)\) | commit |
  | 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 \((A)\) | read \((A)\) |
  | read \((B)\) | write \((A)\) | |
  | write \((A)\) | | |
  | abort | | |

- 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
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 non-serializable 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 serializability help us understand why a concurrency control protocol is correct.

*We’ll cover this next, but first, some final words on transactions*

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.
  - Successive reads of record may return different (but committed) values.
- **Read uncommitted** — even uncommitted records may be read.

Levels of Consistency

- 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 prior to version 9) by default support a level of consistency called snapshot isolation (not part of the SQL standard)
Transaction Definition in SQL

- 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);`
- Isolation level can be set at database level
- Isolation level can be changed at start of transaction
  - E.g. In SQL `set transaction isolation level serializable`
  - E.g. in JDBC -- `connection.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE)`

Transactions as SQL Statements

- E.g., Transaction 1:
  ```sql
  select ID, name from instructor where salary > 90000
  ```
- E.g., Transaction 2:
  ```sql
  insert into instructor values ('11111', 'James', 'Marketing', 100000)
  ```
- Suppose
  - T1 starts, finds tuples salary > 90000 using index and locks them
  - And then T2 executes.
  - Do T1 and T2 conflict? Does tuple level locking detect the conflict?
  - Instance of the **phantom phenomenon**
- Also consider T3 below, with Wu’s salary = 90000
  ```sql
  update instructor
  set salary = salary * 1.1
  where name = 'Wu'
  ```
- Key idea: Detect “predicate” conflicts, and use some form of “predicate locking”