Book Chapters

(4th) Chapters 15, 16.1, 16.7-8, 17.1-4, 17.6
(5th) Chapters 15, 16.1, 16.7-8, 17.1-5
(6th) Chapters 14, 15.1, 15.8-9, 16.1-5

MOTIVATION FOR TRANSACTION

1. Crash recovery
   • ⟨eg, Transfer $1M from Susan to Jane⟩ (example slide)
     – $S_1$: UPDATE Account SET balance = balance - 1000000 WHERE owner = ‘Susan’
     – $S_2$: Update Account SET balance = balance + 1000000 WHERE owner = ‘Jane’
     – System crashes after $S_1$ but before $S_2$. What now?

2. Concurrency
   • We do not want to allow oncurrent access from multiple clients. We do not want to “lock out” the DBMS until one client finishes
     ⟨explain with client/server diagram⟩

   • Can allow parallel execution while avoiding any potential problems from concurrency?
     (we will see concurrency problem examples soon).

TRANSACTION AND “ACID” PROPERTY

• TRANSACTION: A sequence of SQL statements that are executed as a “unit”

• ACID PROPERTY OF TRANSACTION: Atomicity, Consistency, Isolation, Durability

  1. Atomicity: “ALL-OR-NOTHING”
Either ALL OR NONE of the operations in a transaction is executed.

If the system crashes in the middle of a transaction, all changes by the transaction are “undone” during recovery.

2. Durability

After a balance transfer is “done”, the transfer silently “disappears” due to system crash. What will the customer think?

COMMIT: If a transaction “committed”, all its changes remain permanently even after system crash

∗ This guarantee may not be easy because some changes may be reflected only in memory for performance reasons

3. Isolation: Even if multiple transactions are executed concurrently, the result is the same as executing them in some sequential order.

Each transaction is unaware of (is isolated from) other transaction running concurrently in the system

explain by time line diagram

4. Consistency: If the database is in a consistent state before a transaction, the database is in a consistent state after the transaction

• DBMS guarantees the ACID property for all transactions

   With minor caveats that will be discussed later.

• Q: How can the database system guarantee these? Any ideas?

DECLARING A TRANSACTION IN SQL

• Two important commands:

   COMMIT: All changes made by the transaction is stored permanently

   ROLLBACK: Undo all changes made by the transaction

• AUTOCOMMIT MODE

1. With AUTOCOMMIT mode OFF

   Transaction implicitly begins when any data in DB is read or written

   All subsequent read/write is considered to be part of the same transaction

   A transaction finishes when COMMIT or ROLLBACK statement is executed

   explain using time line diagram
2. With AUTOCOMMIT mode ON
   – Every SQL statement becomes one transaction

• Setting Autocommit mode:
  – In DB2: UPDATE COMMAND OPTIONS USING c ON/OFF (default is on)
  – In Oracle: SET AUTOCOMMIT ON/OFF (default is off)
  – In MySQL: SET AUTOCOMMIT = {0|1} (default is on. InnoDB only)
  – In JDBC: connection.setAutoCommit(true/false) (default is on)
  – In Oracle and MySQL, BEGIN temporarily disables autocommit mode until COMMIT or ROLLBACK

TWO QUESTIONS ON TRANSACTION AND CONCURRENCY

1. What execution orders are “good”?
   • We first need to understand what execution orders are okay
     – Serializability theory

2. How can we enforce “good” execution order?
   • Concurrency control mechanism

Topics of next discussion
“GOOD” SCHEDULE

• (ex, salary increase)
(Show example slides)

– \( T_1 \):
  * UPDATE Employee SET salary = salary + 100 WHERE name = ‘Susan’
  * UPDATE Employee SET salary = salary + 100 WHERE name = ‘Jane’

– \( T_2 \):
  * UPDATE Employee SET salary = salary * 2 WHERE name = ‘Susan’
  * UPDATE Employee SET salary = salary * 2 WHERE name = ‘Jane’

Constraint: Susan’s salary = Jane’s salary

Internally, DBMS performs the following operations
A: Susan.salary, B: Jane.salary

– \( T_1 \): Read(A), A=A+100, Write(A), Read(B), B=B+100, Write(B)
– \( T_2 \): Read(A), A=A*2, Write(A), Read(B), B=B*2, Write(B)

• SCHEDULE: The chronological order that instructions are executed in the system
(Show example schedules and show that some schedules are okay and some are not)

– Schedule A:

\[
\begin{array}{c|c}
\text{T1} & \text{T2} \\
Read(A); A &= A+100 \\
Write(A) & \\
Read(B); B &= B+100 \\
Write(B) & Read(A); A &= A*2 \\
Write(A) & \\
Read(B); B &= B*2 \\
Write(B) & \\
\end{array}
\]

– Schedule B:

\[
\begin{array}{c|c}
\text{T1} & \text{T2} \\
& Read(A); A &= A*2 \\
& Write(A) & \\
& Read(B); B &= B*2 \\
& Write(B) & \\
Read(A); A &= A+100 \\
Write(A) & \\
Read(B); B &= B+100 \\
Write(B) & \\
\end{array}
\]
- Schedule C:

<table>
<thead>
<tr>
<th></th>
<th>( T_1 )</th>
<th>( T_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Read(A); ( A = A + 100 )</td>
<td>Read(A); ( A = A \times 2 )</td>
</tr>
<tr>
<td></td>
<td>Write(A)</td>
<td>Write(A)</td>
</tr>
<tr>
<td></td>
<td>Read(B); ( B = B + 100 )</td>
<td>Read(B); ( B = B \times 2 )</td>
</tr>
<tr>
<td></td>
<td>Write(B)</td>
<td>Write(B)</td>
</tr>
</tbody>
</table>

* Q: We get the same or “equivalent” result for schedule A and C. Is it just a coincidence? Is there some reason behind it? (More discussion later...)

- Schedule D:

<table>
<thead>
<tr>
<th></th>
<th>( T_1 )</th>
<th>( T_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Read(A); ( A = A + 100 )</td>
<td>Read(A); ( A = A \times 2 )</td>
</tr>
<tr>
<td></td>
<td>Write(A)</td>
<td>Write(A)</td>
</tr>
<tr>
<td></td>
<td>Read(B); ( B = B + 100 )</td>
<td>Read(B); ( B = B \times 2 )</td>
</tr>
<tr>
<td></td>
<td>Write(B)</td>
<td>Write(B)</td>
</tr>
</tbody>
</table>

* Q: Both C and D executes \( T_1 \) and \( T_2 \) concurrently. Why is one okay but not the other? Any fundamental reason?

- Claim: Two schedules do the “same” thing if
  1. all actions in the two transactions read the same value
  2. they write the same final results to the database

- Intuitively, as long as the transactions read the same values, they will take the same actions. As long as READs and WRITEs are the same transactions do the same thing.

\(<a \text{ simple example}>\)

<table>
<thead>
<tr>
<th>( T_1 )</th>
<th>( T_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( V = r(A) )</td>
<td>( W = r(A) )</td>
</tr>
<tr>
<td>( V = V + 100 )</td>
<td>( W = W + 200 )</td>
</tr>
<tr>
<td>( w(A,V) )</td>
<td>( w(A,W) )</td>
</tr>
</tbody>
</table>

* \( T_1 \) and \( T_2 \) take actions based on input values
  - As long as \( T_1 \) and \( T_2 \) read the same value, \( T_1 \) and \( T_2 \) do the same thing
* We also care about the final output to the database, so the writing to the database is important

- In writing schedules, we just focus on read/write operations
  - What is really important is “read” and “write” actions. All other things depend on these two actions.
  - As long as \( T_1 \) and \( T_2 \) read and write the same values, two schedules are “equivalent”
• Notation for schedule

\[
\begin{array}{c|c}
T_1 & T_2 \\
\hline
V = r(A) & W = r(A) \\
V = V + 100 & W = W + 200 \\
w(A,V) & w(A,W)
\end{array}
\]

\[S: r_1(A) \quad r_2(A) \quad w_2(A) \quad w_1(A)\]

* subscript 1 means transaction 1
* \(r(A)\) means read A
* \(w(A)\) means write to A

**SERIAL SCHEDULE**

• Consider schedule A: \[S_a = \frac{r_1(A) \quad w_1(A) \quad r_1(B) \quad w_1(B)}{T_1} \quad \frac{r_2(A) \quad w_2(A) \quad r_2(B) \quad w_2(B)}{T_2}\]

  – Q: Is this schedule “good”?  

• SERIAL SCHEDULE: all operations in any transaction are performed without any interleaving

  – Definitely a good schedule. The result is definitely good as long as individual transactions are correct.

**CONFLICT SERIALIZABLE SCHEDULE**

• We got the same result for \(S_a\) and \(S_c\). Is there a fundamental reason behind this?

\[S_a = \frac{r_1(A) \quad w_1(A) \quad r_1(B) \quad w_1(B)}{T_1} \quad \frac{r_2(A) \quad w_2(A) \quad r_2(B) \quad w_2(B)}{T_2}\]

\[S_c = \frac{r_1(A) \quad w_1(A)}{T_1} \quad \frac{r_2(A) \quad w_2(A)}{T_2} \quad \frac{r_1(B) \quad w_1(B)}{T_1} \quad \frac{r_2(B) \quad w_2(B)}{T_2}\]

  – Q: Is \(S_c\) a serial schedule?

  – Q: When we swap \(w_2(A)\) and \(r_1(B)\), do we get the “same” result?

\[S_c: S_c = \frac{r_1(A) \quad w_1(A) \quad r_2(A) \quad w_2(A) \quad r_2(B) \quad w_2(B)}{T_1} \quad \frac{r_1(B) \quad w_1(B)}{T_1} \quad \frac{r_2(B) \quad w_2(B)}{T_2}\]

\[S_c': S_c' = \frac{r_1(A) \quad w_1(A) \quad r_2(A) \quad w_2(A)}{T_1} \quad \frac{r_1(B) \quad w_1(B)}{T_2} \quad \frac{r_2(B) \quad w_2(B)}{T_2}\]

• CONFLICTING ACTIONS vs NON-CONFLICTING ACTIONS:
Non-conflicting actions: A pair of actions that do not change the result when we swap them

(Examples)

* Q: \( r_1(A) \ w_2(B) \)? conflicting?

* Q: \( r_1(A) \ r_2(A) \)? conflicting?

* Q: \( w_1(A) \ r_2(A) \)? conflicting?

* Q: \( w_1(A) \ w_2(A) \)? conflicting?

Two actions are CONFLICTING if (1) they involve the same objects/variables and (2) at least one of them is WRITE.

Swapping non-conflicting actions do not change results

Q: Can we swap only non-conflicting actions of \( S_c \) to transform it into \( S_a \)?

* COMMENTS: \( S_a \) and \( S_c \) are guaranteed to produce the same results if we can.

(\{swap actions in \( S_c \) one by one to transform it\})

- CONFLICT EQUIVALENCE

  - DEFINITION: \( S_1 \) is CONFLICT EQUIVALENT to \( S_2 \) if \( S_1 \) can be rearranged into \( S_2 \) by a series of swaps of non-conflicting actions.

  - When \( S_1 \) and \( S_2 \) are conflict equivalent, they read and write same values. Thus, the results are always the same.

- CONFLICT SERIALIZABILITY

  - Intuition:
    * \( S_a \) and \( S_c \) always generate the same results (because they are conflict equivalent)
    * \( S_a \) is guaranteed to generate “good” result because it is SERIAL schedule
    * Thus, \( S_c \) is guaranteed to generate “good” result.

  - FORMAL DEFINITION: \( S_1 \) is CONFLICT SERIALIZABLE if it is conflict equivalent to some serial schedule.

  - A conflict serializable schedule may not be a serial schedule, but it is still a “good” schedule

  - Example:
    * Q:
      \[
      S_1 = r_1(A) \ w_2(A) \ r_1(B) \\
      S_2 = r_1(A) \ r_1(B) \ w_2(A) \text{ conflict equivalent?}
      \]
• Q: What does it mean?
• Q: What does it mean?

∗ Q: $S_1$ conflict serializable?

• Consider schedule D that generates a “wrong” result

Schedule D: $S_c = \frac{r_1(A)}{T1} \frac{w_1(A)}{T1} \frac{r_2(A)}{T2} \frac{w_2(A)}{T2} \frac{r_2(B)}{T2} \frac{w_2(B)}{T2} \frac{r_1(B)}{T1} \frac{w_1(B)}{T1}$

– Q: How should we reorder actions to make it serial?

    ∗ Move the second block of $T_1$ after the first block of $T_1$
    ∗ Move the first block of $T_1$ before the second block of $T_1$

– Q: Can we move $r_1(B)$ before $r_2(A)$ and get the same result?

– Q: Can we move $r_1(A)$ before $r_1(B)$ and get the same result?

– Q: Is Schedule D conflict serializable?

– Q: In general, how can we know whether a schedule is conflict serializable?

– Intuition: Conflicting actions cause precedence relationship between transactions:

    ∗ For example, in schedule D,
        - $w_2(B), \ldots, r_1(B): T_2 \rightarrow T_1$ because $T_1$ reads the value from $T_2$
        - eg. $r_1(A), \ldots, w_2(A): T_1 \rightarrow T_2$ because $T_1$ reads the value before $T_2$

– Q: What does $T_1 \rightarrow T_2$ and $T_2 \rightarrow T_1$ in $S_d$ mean?

– If there is a cycle in the precedence relationship, there is no equivalent serial schedule.
• PRECEDENCE GRAPH P(S)
  - Nodes: transactions in S
  - Edges: $T_i \rightarrow T_j$ if
    1. $p_i(A), q_j(A)$ are actions in S
    2. $p_i(A)$ precedes $q_j(A)$
    3. At least one of $p_i, q_j$ is a write
  - THEOREM: $P(S)$ is acyclic $\iff$ S is conflict serializable
  - $\langle$ eg, $P(S)$ for $S = w_3(A) w_2(C) r_1(A) w_1(B) r_1(C) w_2(A) r_4(A) w_4(D)\rangle$
    * Q: Is S conflict serializable?

  - Q: does $P(S_1) = P(S_2)$ imply $S_1$ and $S_2$ are conflict equivalent?
    * $\langle$ eg, $S_1 = w_1(A) r_2(A) w_2(B) r_1(B)$ and $S_2 = r_2(A) w_1(A) r_1(B) w_2(B)\rangle$

• SUMMARY SO FAR:
  - good schedule: conflict serializable schedule
  - conflict serializable $\iff$ acyclic precedence graph
  - Now more issues related to the meaning of COMMIT

RECOVERABLE SCHEDULE
• (example for recoverable schedule)
  
  $T_1$: $r(A) w(A)$
  $T_2$: $r(A) w(A)$
Can we COMMIT here?

What if ROLLBACK?

- **Q**: Committed transaction may to be aborted! How can we avoid this scenario?

**• RECOVERABLE SCHEDULE**

- Schedule S is RECOVERABLE if \( T_j \) reads a data item written by \( T_i \), the COMMIT operation of \( T_i \) appears before the COMMIT operation of \( T_j \).

- Notation:
  * \( c_i \): COMMIT by transaction \( T_i \)
  * \( a_i \): ROLLBACK by transaction \( T_i \) (abort)

- **Q**: recoverable schedule?

<table>
<thead>
<tr>
<th>( T_1 )</th>
<th>( T_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_1(A) )</td>
<td>( r_2(A) )</td>
</tr>
<tr>
<td>( w_1(A) )</td>
<td>( w_2(A) )</td>
</tr>
<tr>
<td>( c_1 )</td>
<td>( c_2 )</td>
</tr>
</tbody>
</table>

- **Q**: recoverable schedule?

<table>
<thead>
<tr>
<th>( T_1 )</th>
<th>( T_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_1(A) )</td>
<td>( r_2(A) )</td>
</tr>
<tr>
<td>( w_1(A) )</td>
<td>( w_2(A) )</td>
</tr>
<tr>
<td>( c_1 )</td>
<td>( c_2 )</td>
</tr>
</tbody>
</table>

**CASCADELESS SCHEDULE**

- **Q**: What happens to \( T_2 \)?
• CASCADING ROLLBACK: A single transaction abort leads to a series of transaction rollback

  • Q: What is the main cause of cascading rollback?

  – DIRTY READ: reading an output of an uncommitted transaction

  • Q: How can we avoid cascading rollback?

  • CASCADELESS SCHEDULE

  – Schedule S is CASCADELESS if $T_j$ reads a data item written by $T_i$, the COMMIT operation of $T_i$ appears before the READ operation of $T_j$

  – Note the difference between RECOVERABLE and CASCADELESS schedule

RELATIONSHIP BETWEEN SCHEDULES

• Q: What is the relationship between recoverable and cascadeless schedule?
  (Vann diagram)

• Q: What is the relationship between serial and cascadeless schedule?
  (Vann diagram)

• Q: What is the relationship between serial and conflict-serializable schedule?

• Q: What is the relationship between conflict-serializable and recoverable schedule?
– Q: If a schedule is conflict serializable, is it recoverable?

(example)

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w_1(A)$</td>
<td>$r_2(A)$</td>
</tr>
<tr>
<td></td>
<td>$c_2$</td>
</tr>
<tr>
<td>$c_1$</td>
<td></td>
</tr>
</tbody>
</table>

– Q: If a schedule is recoverable, is it conflict serializable?

(example)

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w_1(A)$</td>
<td>$r_2(A)$</td>
</tr>
<tr>
<td>$w_2(A)$</td>
<td>$r_2(A)$</td>
</tr>
<tr>
<td>$c_1$</td>
<td>$c_2$</td>
</tr>
</tbody>
</table>

- **Example S:** $w_1(A)$ $w_2(A)$ $w_1(B)$ $r_2(B)$ $c_1$ $c_2$

  – Q: Is it serial?
  
  – Q: Is it conflict-serializable?

  – Q: Is it recoverable?

  – Q: Is it cascadeless?

  – Q: How can we make it cascadeless?

**WHAT TO REMEMBER**

- Conflict serializable schedule
- Recoverable schedule
- Cascadeless schedule
- We want either
  
  – Conflict serializable + Recoverable
  
  – Conflict serializable + Cascadeless
LOCKING PROTOCOL

• Main question: How can we achieve serializable and cascadeless schedule?
• Let us think how we may achieve serializable and cascadeless schedule.

\[
\begin{array}{c|c}
T_1 & T_2 \\
\hline
r_1(A) & \quad \\ w_1(A) & r_2(A) \\
\hline & w_2(A) \\
\hline
a_1 & \quad \\
\end{array}
\]

– Q: Why do we have cascading rollback? How can we avoid it?

– Q: How can we ensure \( T_2 \) does not read \( T_1 \)'s write until \( T_1 \) commits?

Rigorous Two Phase Locking Protocol (R2PL)

• Basic idea: Avoid dirty read by “locking” modified values until commit.
  1. Before \( T_1 \) writes, \( T_1 \) obtains a lock on A.
  2. \( T_1 \) releases the lock only when \( T_1 \) commits.
  3. When \( T_1 \) holds the lock on A, \( T_2 \) cannot access A.

• Three rules for Rigorous Two Phase Locking Protocol
  – Rule (1): \( T_i \) has to lock tuple \( t_k \) before any read/write
  – Rule (2): When \( T_i \) is holding the lock on \( t_k \), \( T_j \) cannot obtain the lock on \( t_k \) (for \( j \neq i \))
  – Rule (3): Release all locks at commit

(explain using lock accumulation diagram)

• IMPORTANT THEOREM
  – Rigorous 2PL ensures a conflict-serializable and cascadeless schedule.

• RIGOROUS 2PL SCHEDULE
  – A schedule that can be produced by rigorous 2PL protocol
  – Q: Is there any conflict-serializable and cascadeless schedule that cannot be produced by R2PL?
Two Phase Locking Protocol (2PL)

- Less strict locking protocol than rigorous 2PL
  - Rule (1): $T_i$ lock a tuple before any read/write
  - Rule (2): If $T_i$ holds the lock on A, $T_j$ cannot access A ($j \neq i$)
  - Rule (3): Two stages:
    * (a) growing stage: $T_i$ may obtain locks, but may not release any lock
    * (b) shrinking stage: $T_i$ may release locks, but may not obtain any lock

(important theorem)

- 2PL ensures a conflict-serializable schedule

2PL SCHEDULE

- A schedule that can be produced by 2PL protocol.
- Q: Is there any conflict-serializable schedule that cannot be produced by 2PL?

Q: What’s the relationship of R2PL and serializable/cascadless schedule? (explain using Vann Diagram)

Q: What’s the relationship of 2PL and serializable/cascadless schedule? (explain using Vann Diagram)
RECOVERY AND LOGGING

• Motivation for logging. Consider $T : r(A)w(A)r(B)w(B)$.

  – **Example 1**: $S = r(A)w(A)r(B)a$. What should we do? How do we get the old value of $A$?

  – **Example 2**: $S = r(A)w(A)$ !!!CRASH!!! What should DBMS do when it reboots?

  – **Example 3**: $S = r(A)w(A)r(B)w(B)c$. New $A$ and $B$ values are “cached” in main memory for performance reasons. Can DBMS commit $T$ without writing the new values permanently to the disk? (main-memory and disk diagram)

• Rules for log-based recovery

  1. For every action DBMS performs, a “log record” for the action should be generated.

     – $\langle T_i, \text{start} \rangle$
     – $\langle T_i, X_j, \text{old-value, new-value} \rangle$
     – $\langle T_i, \text{commit} \rangle$
     – $\langle T_i, \text{abort} \rangle$

  2. Log record should be written to disk BEFORE the actual data is written to the disk.

     – $\langle T_i, A, 5, 10 \rangle$ should be written before the new $A$ value 10 is written to the data block.

  3. Before commit of $T_i$, all log records for $T_i$ are written to disk (including commit).

     – The actual data block may or may not be written to disk at commit.

  4. During abort, DBMS gets old values from the log

  5. During recovery, DBMS does the following:

     (a) “re-executes” all actions in the log from the beginning.

     (b) “rolls back” all actions of “non-committed” transactions in the reverse order.

• We can use CHECKPOINT to minimize recovery time.
**Example:**
(Explain log records line by line)

A: 100, B: 100, C: 100

<table>
<thead>
<tr>
<th>( T_1 )</th>
<th>( T_2 )</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A); A=A-50</td>
<td>Read(C); C=C*2</td>
<td>1 ( \langle T_1, \text{start} \rangle )</td>
</tr>
<tr>
<td>Write(A)</td>
<td>Write(C)</td>
<td>2 ( \langle T_1, A, 100, 50 \rangle )</td>
</tr>
<tr>
<td>Read(B); B=B+50</td>
<td>Commit</td>
<td>3 ( \langle T_2, \text{start} \rangle )</td>
</tr>
<tr>
<td>Write(B)</td>
<td></td>
<td>4 ( \langle T_2, C, 100, 200 \rangle )</td>
</tr>
<tr>
<td>Commit</td>
<td></td>
<td>5 ( \langle T_2, \text{commit} \rangle )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6 ( \langle T_1, B, 100, 150 \rangle )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7 ( \langle T_1, \text{commit} \rangle )</td>
</tr>
</tbody>
</table>

- **Q:** What should DBMS do during recovery when it sees up to log record 4?

- **Q:** What should DBMS do during recovery when it sees up to log record 5?

- **Q:** What should DBMS do during recovery when it sees up to log record 7?
SQL ISOLATION LEVELS

• Motivation: In some cases, we may not need full ACID. We may want to allow some “bad” schedule to achieve more concurrency
  – SQL isolation levels allow a few “bad” scenarios for more concurrency
    * dirty read, non-repeatable read, phantom
  – We go over three scenarios in which “relaxing” the strict ACID may be desirable for some applications

• (explain the isolation levels through examples and fill in the table)

<table>
<thead>
<tr>
<th>isolation level</th>
<th>dirty read</th>
<th>nonrepeatable read</th>
<th>phantom</th>
</tr>
</thead>
<tbody>
<tr>
<td>read uncommitted</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>read committed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>repeatable read</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>serializable</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• DIRTY READ may be OK
  – (example)
    * \( T_1 \): UPDATE Employee SET salary = salary + 100
    * \( T_2 \): SELECT salary FROM Employee WHERE name = ‘John’
  – Q: Under ACID, once \( T_1 \) update John’s salary, can \( T_2 \) read John’s salary?

    * Sometimes, it may be okay for \( T_2 \) to proceed.
  – DIRTY READ: a transaction reads uncommitted values
  – “READ UNCOMMITTED” isolation level allows dirty read.
    (Fill in the dirty read column)

• NON-REPEATABLE READ may be OK
  – (example)
    * \( T_1 \): UPDATE Employee SET salary = salary + 100 WHERE name = ‘John’
    * \( T_2 \): (\( S_1 \)) SELECT salary FROM Employee WHERE name = ‘John’
    . . .
    (\( S_2 \)) SELECT salary FROM Employee WHERE name = ‘John’
  – Q: Under ACID, can we get different values for \( S_1 \) and \( S_2 \)?

    * Sometimes it may be okay to get different values
  – NON-REPEATABLE READ: When \( T_i \) reads the same row multiple times, \( T_i \) may get different values
  – “READ UNCOMMITTED” or “READ COMMITTED” isolation levels allow NON-REPEATABLE READ.
    (Fill in the non-repeatable read column)
• PHANTOM may be OK
  
  *(example)*
  
  * Initially, \( \text{SUM}(\text{Employee}.\text{salary}) = $100,000 \)*
  * \( T_1 \): \text{INSERT INTO Employee (e1, 1000), (e2, 1000) }\)*
  * \( T_2 \): \text{SELECT SUM(salary) FROM Employee }\)*
  
  – **Q:** Under ACID, what may \( T_2 \) return?

  * Sometimes, it may be OK for \( T_2 \) to return $101,000

  – **Q:** Under REPEATABLE READ, what if \( T_2 \) is

  \[
  \text{SELECT SUM(salary) FROM Employee }\]

  \[
  \text{...}
  \]

  \[
  \text{SELECT SUM(salary) FROM Employee }\]

  What can \( T_2 \) return?

  – PHANTOM: When new tuples are inserted, once some of them are seen by statements, or only some statements see the newly inserted tuples.
  – Except for “SERIALIZABLE” isolation level, PHANTOM is always allowed.

• MIXED ISOLATION LEVELS

  *(example on mixed isolation levels)*

  * \( T_1 \): \text{UPDATE Employee SET salary = salary + 100 }\text{ROLLBACK}*
  
  * \( T_2 \): \text{SELECT salary FROM Employee WHERE name = ‘John’ }\)*
  
  – **Q:** \( T_1 \) - SERIALIZABLE, \( T_2 \) - SERIALIZABLE. What may \( T_2 \) return?

  – **Q:** \( T_1 \) - SERIALIZABLE, \( T_2 \) - READ UNCOMMITTED. What may \( T_2 \) return?

  – **COMMENTS:**
    * Only when all transactions are serializable, we guarantee ACID.
    * The isolation level is in the eye of the beholding transaction.

• READ ONLY TRANSACTION
Many, many transactions are read only.
By declaring a transaction as READ ONLY, we can help DBMS to optimize for more concurrency

• SQL ISOLATION LEVEL DECLARATION

  • SET TRANSACTION options
  • access mode: READ ONLY / READ WRITE (default: READ WRITE)
  • isolation level: ISOLATION LEVEL
    * READ UNCOMMITTED
    * READ COMMITTED (Oracle default)
    * REPEATABLE READ (MySQL, DB2 default)
    * SERIALIZABLE
  • e.g) SET TRANSACTION READ ONLY, ISOLATION LEVEL REPEATABLE READ
    * READ UNCOMMITTED cannot be READ WRITE
    * Needs to be declared before EVERY transaction for non-default settings