Quiz 3

• Query Processing (only the last half and join cost models)
• Transactions
• Concurrency Control
• Recovery
• Distributed Databases (only a light question)
Query Processing
Cardinality of Joins in general

Assume join: \( R \bowtie S \) (common attributes are not keys)

1. If \( R, S \) have no common attributes: \( n_r \cdot n_s \)
2. If \( R, S \) have attribute \( A \) in common:

\[
\frac{n_r \cdot n_s}{V(A,s)} \text{ or } \frac{n_s \cdot n_r}{V(A,r)} \quad \text{(take min)}
\]

- These are not the same when \( V(A,s) \neq V(A,r) \).
- When this is true, there are likely to be dangling tuples.
- Thus, the smaller is likely to be more accurate.
Nested-Loop Join

Query: \[ R \Join S \]

Algorithm 1: Nested Loop Join

for each tuple \( t_r \) in \( R \) do
  for each tuple \( u_s \) in \( S \) do
    test pair \((t_r,u_s)\) to see if they satisfy the join condition
    if they do (a “match”), add \( t_r \cdot u_s \) to the result.

\( R \) is called the outer relation and \( S \) the inner relation of the join.

Nested-Loop Join (Cont.)

Cost:
- Worst case, if buffer size is 3 blocks
  \[ b_r + n_r \cdot b_s \] disk accesses.
- Best case: buffer big enough for entire INNER relation + 2
  \[ b_r + b_s \] disk accesses.
Block Nested-Loop Join

- Block Nested Loop Join
  
  for each block $B_R$ of $R$ do
  
  for each block $B_S$ of $S$ do
  
  for each tuple $t_r$ in $B_R$ do
  
  for each tuple $u_s$ in $B_S$ do begin
  
  Check if $(t_r, u_s)$ satisfy the join condition
  
  if they do (“match”), add $t_r \cdot u_s$ to the result.

Block Nested-Loop Join (Cont.)

Cost:

- Worst case estimate (3 blocks): $b_r \cdot b_s + b_r$ block accesses.

- Best case: $b_r + b_s$ block accesses. Same as nested loop.
Indexed Nested-Loop Join

Indexed Nested Loop Join

- For each tuple $t_R$ in the outer relation $R$, use the index to look up tuples in $S$ that satisfy the join condition with tuple $t_R$.

- Worst case: buffer has space for only one page of $R$, and, for each tuple in $R$, we perform an index lookup on $S$.

- Cost of the join: $b_r + n_r * c$
  - Where $c$ is the cost of traversing index and fetching all matching $s$ tuples for one tuple of $r$
  - $c$ can be estimated as cost of a single selection on $s$ using the join condition.

- If indices are available on join attributes of both $R$ and $S$, use the relation with fewer tuples as the outer relation.
Merge-Join

Idea: suppose R, S are both sorted on A (A is the common attribute)

<table>
<thead>
<tr>
<th>A</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>p_R</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

... p_R

<table>
<thead>
<tr>
<th>A</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>p_S</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

... p_S

Compare:
(1, 2) advance p_R
(2, 2) match, advance p_S → add to result
(2, 2) match, advance p_S → add to result
(2, 3) advance p_R
(3, 3) match, advance p_S → add to result
(3, 5) advance p_R
(4, 5) read next block of R

Cost: $b_R + b_S + \text{sort}_R + \text{sort}_S$
External Sorting (cont.)

Cost: \[ 2 b_R \times (\lceil \log_{M-1}(b_R / M) \rceil + 1) \]

Step 1: Create runs
- every block read and written once
- cost = \( 2 b_R \) I/Os

Number of iterations

Step 2: Merge
- every merge iteration requires reading and writing entire file (\( = 2 b_R \) I/Os)
- every iteration reduces the number of runs by factor of \( M-1 \)

<table>
<thead>
<tr>
<th>Iteration #</th>
<th>Runs Left to Merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \frac{b_R}{M} )</td>
</tr>
<tr>
<td>2</td>
<td>( \frac{b_R}{M} \frac{1}{M-1} )</td>
</tr>
<tr>
<td>3</td>
<td>( \frac{b_R}{M} \frac{1}{M(M-1)} )</td>
</tr>
<tr>
<td>...</td>
<td>( \frac{b_R}{M} \frac{1}{M(M-1)} \frac{1}{M-1} )</td>
</tr>
</tbody>
</table>

Initial number of runs

\# merge passes = \( \lceil \log_{M-1}(b_R / M) \rceil \)
Hash-Join Algorithm

The hash-join of \( r \) and \( s \) is computed as follows.

1. Partition the relation \( s \) using hashing function \( h \). When partitioning a relation, one block of memory is reserved as the output buffer for each partition.

2. Partition \( r \) similarly.

3. For each \( i \):
   
   (a) Load \( s_i \) into memory and build an in-memory hash index on it using the join attribute. This hash index uses a different hash function than the earlier one \( h \).

   (b) Read the tuples in \( r_i \) from the disk one by one. For each tuple \( t_i \) locate each matching tuple \( t_s \) in \( s_i \) using the in-memory hash index. Output the concatenation of their attributes.

Cost of hash join is

\[ 3(b_r + b_s) + 4 \times n_h \text{ block transfers} \]

- If the entire build input can be kept in main memory no partitioning is required
  
  ★ Cost estimate goes down to \( b_r + b_s \).
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
Updates in SQL

An example:

UPDATE account
SET balance = balance - 50
WHERE acct_no = A102

Transaction:
1. Read(A)
3. Write(A)

What takes place:

Disk

(1) Read
(2) update
(3) write
The Threat to Data Integrity

Consistent DB
Name Acct bal
------- ------ ------
Joe A-102 300
Joe A-509 100
Joe's total: 400

Inconsistent DB
Name Acct bal
------- ------ ------
Joe A-102 250
Joe A-509 100
Joe's total: 350

Move $50 from acct A-102 to acct A-509

Consistent DB
Name Acct bal
------- ------ ------
Joe A-102 250
Joe A-509 150
Joe's total: 400

What a Xaction should look like to Joe

What actually happens during execution
ACID Properties

Properties that a Xaction needs to have:

- **Atomicity**: either all operations in a Xaction take effect, or none

- **Consistency**: operations, taken together preserve db consistency

- **Isolation**: intermediate, inconsistent states must be concealed from other Xactions

- **Durability**: If a Xaction successfully completes ("commits"), changes made to db must persist, even if system crashes
Demonstrating ACID

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)

\[ \text{FAILURE!} \]

Consistency: total value A+B, unchanged by Xaction

Atomicity: if Xaction fails after 3 and before 6, 3 should not affect db

Durability: once user notified of Xaction commit, updates to A,B should not be undone by system failure

Isolation: other Xactions should not be able to see A, B between steps 3-6
Schedules

- Schedules – sequences that indicate 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

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>D</td>
</tr>
</tbody>
</table>

one possible schedule:

```
T1  T2
1    A
2    B
3    C
```

```
T1  T2
1    A
    B
2    C
    D
```
Example Schedules

Transactions:
T1: transfers $50 from A to B
T2: transfers 10% of A to B

Example 1: a "serial" schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td></td>
</tr>
<tr>
<td>A &lt;- A -50</td>
<td></td>
</tr>
<tr>
<td>write(A)</td>
<td></td>
</tr>
<tr>
<td>read(B)</td>
<td></td>
</tr>
<tr>
<td>B &lt;- B +50</td>
<td></td>
</tr>
<tr>
<td>write(B)</td>
<td></td>
</tr>
</tbody>
</table>

Constraint: The sum of A+B must be the same

Before: 100 + 50
= 150, consistent

After: 45 + 105
Conflict Serializability (Cont.)

- 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.

- Ex:

```
T1  T2
......  ......  
read(A)  read(A)  can be rewritten to equivalent schedule

T1  T2
......  ......  
read(A)  read(A)
```

Conflict Serializability (Cont.)

Example:

T1
1. Read(A)
2. A ← A - 50
3. Write(A)

T2
4. Read(B)
5. B ← B + 50
6. Write(B)

Swaps:

| 4 <->d | 5 <->d | 6 <->d |
| 4 <->c | 5 <->c | 6 <->c |
| 4 <->b | 5 <->b | 6 <->b |
| 4 <->a | 5 <->a | 6 <->a |

Conflict serializable

T1, T2
Conflict Serializability (Cont.)

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td></td>
</tr>
<tr>
<td>A &lt;- A -50</td>
<td></td>
</tr>
<tr>
<td>write(A)</td>
<td></td>
</tr>
<tr>
<td>read(B)</td>
<td></td>
</tr>
<tr>
<td>B &lt;- B+50</td>
<td></td>
</tr>
<tr>
<td>write(B)</td>
<td></td>
</tr>
</tbody>
</table>

The effects of swaps

Because example schedule could be swapped to this schedule (<T1, T2>)

Example schedule is conflict serializable

read(A)
tmp <- A * 0.1
A <- A - tmp
write(A)
read(B)
B <- B + tmp
write(B)
Conflict Serializability (Cont.)

Previous example w.o. local operations:

T1
1. Read(A)
2. Write(A)
3. Read(B)
4. Write(B)

T2
a. Read(A)
b. Write(A)
c. Read(B)
d. Write(B)

Swaps:
3 <-> b
3 <-> a
4 <-> b
4 <-> a
T1, T2
Swappable Operations

- Swappable operations:
  1. Any operation on different data element
  2. Reads of the same data (Read(A))
     (regardless of order of reads, the same value for A is read)

- Conflicts:

<table>
<thead>
<tr>
<th></th>
<th>T2: Read(A)</th>
<th>T2: Write(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1: Read (A)</td>
<td>OK</td>
<td>R/W Conflict</td>
</tr>
<tr>
<td>T1: Write (A)</td>
<td>W/R Conflict</td>
<td>W/W Conflict</td>
</tr>
</tbody>
</table>
Conflict Serializability

Testing: too expensive to test a schedule by swapping operations (usually schedules are big!)

Alternative: “Precedence Graphs”
  * vertices = Xactions
  * edges = conflicts between Xactions

E.g.: $T_i \rightarrow T_j$ if: (1) $T_i, T_j$ have a conflicting operation, and (2) $T_i$ executed its conflicting operation first
Another example:

Not conflict serializable!!
Because there is a cycle in the PG, the cycle creates contradiction
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.

For example, a serializability order for Schedule A would be $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$. 
Other things to put on your cheat sheet...

• View Serializability (solely based on R/W) 3 conditions.
• Recoverable schedules
• Cascading rollbacks (one transaction failures causes a series of transactions to be undone)
• Cascadeless schedules
**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 Via Locks

- Idea:
  - Data items modified by one transaction at a time

- Locks
  - Control access to a resource
    - Can block a transaction until lock granted
  - Two modes:
    - Shared (read only)
    - eXclusive (read & write)
# Lock instructions

## New instructions
- `lock-S`: shared lock request
- `lock-X`: exclusive lock request
- `unlock`: release previously held lock

### Example:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lock-X(B)</code></td>
<td><code>lock-S(A)</code></td>
</tr>
<tr>
<td><code>read(B)</code></td>
<td><code>read(A)</code></td>
</tr>
<tr>
<td><code>B ← B-50</code></td>
<td><code>unlock(A)</code></td>
</tr>
<tr>
<td><code>write(B)</code></td>
<td><code>lock-S(B)</code></td>
</tr>
<tr>
<td><code>unlock(B)</code></td>
<td><code>read(B)</code></td>
</tr>
<tr>
<td><code>lock-X(A)</code></td>
<td><code>unlock(B)</code></td>
</tr>
<tr>
<td><code>read(A)</code></td>
<td><code>display(A+B)</code></td>
</tr>
<tr>
<td><code>A ← A + 50</code></td>
<td></td>
</tr>
<tr>
<td><code>write(A)</code></td>
<td></td>
</tr>
<tr>
<td><code>unlock(A)</code></td>
<td></td>
</tr>
</tbody>
</table>
Locking Issues

- Starvation
  - T1 holds shared lock on Q
  - T2 requests exclusive lock on Q: blocks
  - T3, T4, ..., Tn request shared locks: granted
  - T2 is starved!

- Solution?

  Do not grant locks if older transaction is waiting
**Locking Issues**

- No transaction proceeds:
  - Deadlock
    - T1 waits for T2 to unlock A
    - T2 waits for T1 to unlock B

Rollback transactions
Can be costly...

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-X(A)</td>
<td>lock-S(B)</td>
<td>read(A)</td>
</tr>
<tr>
<td>read(B)</td>
<td>write(B)</td>
<td>B ← B-50</td>
</tr>
<tr>
<td>lock-X(B)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Locking Issues

- Locks do not ensure serializability by themselves:

T1
lock-X(B)
read(B)
B ← B - 50
write(B)
unlock(B)

T2 displays 50 less!!

T2
lock-S(A)
read(A)
unlock(A)
lock-S(B)
read(B)
unlock(B)
display(A + B)

lock-X(A)
read(A)
A ← A + 50
write(A)
unlock(A)
The Two-Phase Locking Protocol

- This is a protocol which ensures conflict-serializable schedules.
- Phase 1: Growing Phase
  - transaction may obtain locks
  - transaction may not release locks
- Phase 2: Shrinking Phase
  - transaction may release locks
  - transaction may not obtain locks
- The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their lock points (i.e. the point where a transaction acquired its final lock). Locks can be either X, or S/X.
Example: T1 in 2PL

Growing phase

Shrinking phase

<table>
<thead>
<tr>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-X(B)</td>
</tr>
<tr>
<td>read(B)</td>
</tr>
<tr>
<td>B ← B - 50</td>
</tr>
<tr>
<td>write(B)</td>
</tr>
<tr>
<td>lock-X(A)</td>
</tr>
<tr>
<td>read(A)</td>
</tr>
<tr>
<td>A ← A - 50</td>
</tr>
<tr>
<td>write(A)</td>
</tr>
<tr>
<td>unlock(B)</td>
</tr>
<tr>
<td>unlock(A)</td>
</tr>
</tbody>
</table>
2PL Issues

- As observed earlier, 2PL does not prevent deadlock
- > 2 transactions involved?
  - Rollbacks expensive.
- We will revisit later.
2PL Variants

Strict two phase locking
- Exclusive locks must be held until transaction commits
- Ensures data written by transaction can’t be read by others
- Prevents cascading rollbacks
### Strict 2PL

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-X(A)</td>
<td>lock-X(A)</td>
<td>lock-S(A)</td>
</tr>
<tr>
<td>read(A)</td>
<td>read(A)</td>
<td>read(A)</td>
</tr>
<tr>
<td>lock-S(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unlock(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>lock-S(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>read(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;xaction fails&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Strict 2PL will not allow that.
Deadlock Handling

- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.

- **Deadlock prevention** protocols ensure that the system will never enter into a deadlock state. Some prevention strategies:
  - Require that each transaction locks all its data items before it begins execution (predeclaration).
  - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol).
More Deadlock Prevention Strategies

- Following schemes use transaction timestamps for the sake of deadlock prevention alone.

- **wait-die** scheme — non-preemptive
  - older transaction may wait for younger one to release data item. Younger transactions never wait for older ones; they are rolled back instead.
  - a transaction may die several times before acquiring needed data item

- **wound-wait** scheme — preemptive
  - older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.
Deadlock Prevention

<table>
<thead>
<tr>
<th></th>
<th>Wait / Die</th>
<th>Wound / Wait</th>
</tr>
</thead>
<tbody>
<tr>
<td>O Needs a resource held by Y</td>
<td>O Waits</td>
<td>Y Dies</td>
</tr>
<tr>
<td>Y needs a resource held by O</td>
<td>Y Dies</td>
<td>Y Waits</td>
</tr>
</tbody>
</table>

WAIT
Req by Old
Locked by Young

DIE
Req by Young
Locked by Old

WAIT
Req by Young
Locked by Old

WOUND
Locked by Young

WAIT / DIE

WOUND / WAIT
Dealing with Deadlocks

- How do you detect a deadlock?
  - Wait-for graph
  - Directed edge from Ti to Tj
    - If Ti waiting for Tj

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td>S(V)</td>
<td>X(V)</td>
<td>X(Z)</td>
<td>X(W)</td>
</tr>
<tr>
<td>S(W)</td>
<td>S(V)</td>
<td>S(V)</td>
<td>S(V)</td>
</tr>
</tbody>
</table>

Suppose T4 requests lock-S(Z)....
Detecting Deadlocks

- Wait-for graph has a cycle $\rightarrow$ deadlock

T2, T3, T4 are deadlocked

- Build wait-for graph, check for cycle

- How often?
  - Tunable
    - IF expect many deadlocks or many transactions involved
    - run often to reduce aborts
    - ELSE run less often to reduce overhead
Recovering from Deadlocks

- Rollback one or more transactions
  - Which one?
    - Rollback the cheapest ones
    - Cheapest ill-defined
      - Was it almost done?
      - How much will it have to redo?
      - Will it cause other rollbacks?
  - How far?
    - May only need a partial rollback
  - Avoid starvation
    - Ensure same xction not always chosen to break deadlock
Timestamp-Based Protocols

- Idea:
  - Decide in advance ordering of transactions.
  - Ensure concurrent schedule serializes to that serial order.

- Timestamps
  1. $TS(T_i)$ is time $T_i$ entered the system
  2. Data item timestamps:
     1. $W-TS(Q)$: Largest timestamp of any xction that wrote $Q$
     2. $R-TS(Q)$: Largest timestamp of any xction that read $Q$

- Timestamps $\rightarrow$ serializability order
Timestamp CC

**Idea:** If action $p_i$ of Xact $T_i$ conflicts with action $q_j$ of Xact $T_j$, and $TS(T_i) < TS(T_j)$, then $p_i$ must occur before $q_j$. Otherwise, restart violating Xact.
When Xact T wants to read Object O

- If $TS(T) < W-TS(O)$, this violates timestamp order of T w.r.t. writer of O.
  - So, abort T and restart it with a new, larger TS. (If restarted with same TS, T will fail again!)

- If $TS(T) > W-TS(O)$:
  - Allow T to read O.
  - Reset R-TS(O) to max(R-TS(O), TS(T))

- Change to R-TS(O) on reads must be written to disk! This and restarts represent overhead.
When Xact T wants to Write Object O

- If $TS(T) < R-TS(O)$, then the value of $O$ that $T$ is producing was needed previously, and the system assumed that that value would never be produced. **write rejected, $T$ is rolled back.**

- If $TS(T) < W-TS(O)$, then $T$ is attempting to write an obsolete value of $O$. Hence, this **write operation is rejected, and $T$ is rolled back.**

- Otherwise, the **write operation is executed, and W-TS(O) is set to TS(T).**
Example Use of the Protocol

A partial schedule for several data items for transactions with initial timestamps 1, 2, 3, 4, 5

<table>
<thead>
<tr>
<th></th>
<th>$T_2$</th>
<th>$T_3$</th>
<th>$T_4$</th>
<th>$T_5$</th>
<th>$T_6$</th>
<th>$T_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>read($Y$)</td>
<td>read($Y$)</td>
<td>write($Y$)</td>
<td>write($X$)</td>
<td>read($Y$)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>read($X$)</td>
<td></td>
<td>write($Y$)</td>
<td>write($Z$)</td>
<td>read($Z$)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>abort</td>
<td></td>
<td>abort</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_2$</td>
<td></td>
<td></td>
<td></td>
<td>read($Z$)</td>
<td>write($Y$)</td>
<td>read($X$)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_5$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_6$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_7$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Recovery
Review: The ACID properties

Atomicity: All actions in the Xaction happen, or none happen.

Consistency: If each Xaction is consistent, and the DB starts consistent, it ends up consistent.

Isolation: Execution of one Xaction is isolated from that of other Xacts.

Durability: If a Xaction commits, its effects persist.

CC guarantees Isolation and Consistency.
The Recovery Manager guarantees Atomicity & Durability.
Recovery and Atomicity (Cont.)

To ensure atomicity, first output information about modifications to stable storage without modifying the database itself.

We study two approaches:
  - log-based recovery, and
  - shadow-paging
Log-Based Recovery

Log: a sequence of log records; maintains a record of update activities on the database. (Write Ahead Log, W.A.L.)

Log records for transaction Ti:

\(<Ti \text{ start } >\)
\(<Ti , X , V1, V2>\)
\(<Ti \text{ commit } >\)

Two approaches using logs

Deferred database modification
Immediate database modification
Deferred Database Modification

- Ti starts: write a `<Ti start>` record to log.
- Ti `write(X)`
  - write `<Ti, X, V> to log: V is the new value for X`
  - The write is deferred
  - Note: old value is not needed for this scheme
- Ti partially commits:
  - Write `<Ti commit>` to the log
- DB updates by reading and executing the log:
  - `<Ti start> ...... <Ti commit>`

How to use the log for recovery after a crash?
Redo: if both `<Ti start>` and `<Ti commit>` are there in the log.
Immediate Database Modification

Tighter logging rules are needed to ensure transactions are undoable

LOG records must be of the form: \(<\textit{Ti}, \textit{X}, \textit{Vold}, \textit{Vnew}\)>

Log record must be written \textit{before} database item is written

Immediate Database Modification (Cont.)

Recovery procedure:

\textbf{Undo}: \(<\textit{Ti}, \textit{start}\) is in the log but \(<\textit{Ti commit}\) is not. Undo:
restore the value of all data items updated by \textit{Ti} to their old values, going backwards from the last log record for \textit{Ti}

\textbf{Redo}: \(<\textit{Ti start}\) and \(<\textit{Ti commit}\) are both in the log.
sets the value of all data items updated by \textit{Ti} to the new values, going forward from the first log record for \textit{Ti}
Checkpoints

Problems in recovery procedure as discussed earlier:
- searching the entire log is time-consuming
- we might unnecessarily redo transactions which have already output their updates to the database.

How to avoid redundant redoes?
- Put marks in the log indicating that at that point DB and log are consistent. **Checkpoint!**

**Recovering from log with checkpoints:**

1. Scan backwards from end of log to find the most recent `<checkpoint>` record

2. Continue scanning backwards till a record `<Ti start>` is found.

3. Need only consider the part of log following above `start` record. Why?

4. After that, recover from log with the rules that we had before.
Recovery With Concurrent Transactions (Cont.)

Checkpoints for concurrent transactions:

<checkpoint L>

L: the list of transactions active at the time of the checkpoint

We assume no updates are in progress while the checkpoint is carried out

Recovery for concurrent transactions, 3 phases:

Initialize undo-list and redo-list to empty

Scan the log backwards from the end, stopping when the first <checkpoint L> record is found.

For each record found during the backward scan:

- if the record is <Ti commit>, add Ti to redo-list
- if the record is <Ti start>, then if Ti is not in redo-list, add Ti to undo-list

For every Ti in L, if Ti is not in redo-list, add Ti to undo-list
Recovery With Concurrent Transactions

Scan log backwards
   Perform undo(T) for every transaction in undo-list
   Stop when you have seen <T, start> for every T in undo-list.

Locate the most recent <checkpoint L> record.
   Scan log forwards from the <checkpoint L> record till the end of the log.
   perform redo for each log record that belongs to a transaction on redo-list

UNDO

REDO