Concurrent Processing and Recovery

(only for DBs with updates.....!)

- Concurrency Control
  - Transaction
  - ACID
  - Isolation
    - Schedules
    - Guaranteeing isolation
      - Serializability
      - Conflict Serializability
      - Precedence Graph
      - Locking
      - Two Phase Locking
      - Deadlock
      - Isolation levels

- Recovery
  - ACID: Atomicity & Durability
  - Trading execution time for recovery time
  - Mechanics of recovery
    - What is recovered and how
    - Write ahead logging
    - Mechanics of logging
      - Log Record Contents
      - How to commit, abort and redo
    - Big Picture

Why Have Concurrent Processing (from multiple users) against a Database?

- Increase throughput, response time

- Better utilization of resources: While one processes is doing a disk read, another can be using the CPU or reading another disk.

- But...concurrent processing can lead to incorrect data in the database or incorrect data returned to users!
Transaction

A transaction is:
- one “complete” set of actions
- defined by the user (meaningful to the user)
- establishes where certain integrity constraints are enforced.

For concurrency control purposes (inside DBMS):
- a transaction is one atomic unit of work.
- DBMS cares about the reads/writes to the DB
- DBMS views a transaction as (only) a sequence of reads, writes plus commit & abort (ignoring the rest of the program)

Example Transaction

Transfer $100 from one account to another:
read balance from first account
check balance to see if there’s enough money
read balance from second account
subtract $100 from first account balance
add $100 to second account balance
update first account record
update second account record
print receipt
Example Transaction

Transfer $100 from one account to another:
read balance from first account
check balance to see if there is enough money
read balance from second account
subtract $100 from first account balance
add $100 to second account balance
update first account record
update second account record
print receipt

Example of a (VERY LONG) Transaction
(probably one that you do not want to have..)

Reduce the price of all products in the shoe department by 5%
read product1
adjust price
modify product1 record
read product2
adjust price
modify product2 record

...
Transaction (cont.)

User (application developer) must indicate:
- Begin transaction
- read/write/modify statements intermixed with other programming language statements

plus either
- commit - indicates successful completion or
- abort - indicates program wants to roll back (erase the transaction)

The ACID Properties of Transactions

- **Atomicity**: All actions in the transaction happen in their entirety or not at all.
- **Consistency**: If each transaction is consistent, and the DB starts consistent, the DB ends up consistent.
- **Isolation**: The transaction is isolated from other transactions. The effect on the DB is (as if) the transaction executed by itself.
- **Durability**: If a transaction commits, its changes to the database state persist (changes are permanent).
Supporting the ACID Properties of Transactions

**Atomicity:** All actions in the transaction happen in their entirety or not at all.

**Consistency:** If each transaction is consistent, and the DB starts consistent, the DB ends up consistent. Developers must make the transactions consistent.

**Isolation:** The transaction is isolated from other transactions. The effect on the DB is (as if) the transaction executed by itself.

**Durability:** If a transaction commits, its changes to the database state persist.

---

Example: Isolation

- Consider two transactions:

  T1: BEGIN A=A+100, B=B-100 END
  T2: BEGIN A=1.06*A, B=1.06*B END

- Intuitively, the first transaction is transferring $100 from B’s account to A’s account. The second is crediting both accounts with a 6% interest payment.
Possible schedules for T1 and T2

<table>
<thead>
<tr>
<th>T2 then T1</th>
<th>T1 then T2</th>
<th>T2 between T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>1000</td>
<td>500</td>
<td>1000</td>
</tr>
<tr>
<td>1060</td>
<td>1100</td>
<td>1100</td>
</tr>
<tr>
<td>1160</td>
<td>430</td>
<td>1166</td>
</tr>
<tr>
<td>1166</td>
<td>424</td>
<td>430</td>
</tr>
</tbody>
</table>

Initial values for A, B

Legal values for A and B
(either order of T1 & T2 is correct from a DBMS point of view)

Illegal values!

Interleaved Schedules:

Consider this interleaved schedule:

T1: A=A+100, B=B-100
T2: A=1.06*A, B=1.06*B

Equivalent to T1 before T2 on prior page. This is OK.

Consider another interleaved schedule:

T1: A=A+100, B=B-100
T2: A=1.06*A, B=1.06*B

Equivalent to third schedule on prior page. This is illegal.
Schedules

- **Schedule:** An interleaving of actions from a set of transactions, where the actions of any 1 transaction are in the original order.
  - Represents an actual sequence of database actions.
  - Example: R₁(A), W₁(A), R₂(B), W₂(B), R₁(C), W₁(C)
  - In a complete schedule, each transaction ends in commit or abort.

- **Initial State + Schedule → Final State**

Acceptable (Isolated) Schedules

(Acceptable ≡ Correct ≡ Isolated)

- **Serial** schedules:
  - Run transactions one at a time, in a series.

- **Serializable** schedules:
  - Final state must be the same as the state produced by one of the serial schedules.
  - Aborted transactions are not part of schedule; ignore them for now (they are made to ‘disappear’ by using recovery subsystem).
Examples: Serializable Schedules

Which of these schedules is serializable?

<table>
<thead>
<tr>
<th></th>
<th>T1: R(A), W(A)</th>
<th>R(B), W(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2:</td>
<td>R(A), W(A)</td>
<td>R(B), W(B)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>T1: R(A), W(A)</th>
<th>R(B), W(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2:</td>
<td>R(A), W(A), R(B), W(B)</td>
<td></td>
</tr>
</tbody>
</table>

Conflicts in Concurrent Transactions

We have a conflict if:

different transactions, operate on the same object, and one of the operations is a write


Three Kinds of Conflicts

- **Write-Read conflicts (dirty reads)**
  - Some transaction modifies data
  - Then you read it
  - Then the other transaction aborts – and their modifications to the database are UNDONE
  - You have read a tuple value that should never have existed.

- **Read-Write conflicts (unrepeatable reads)**
  - You read a value
  - Then some other transaction modifies it
  - Then you read it again (IT’S DIFFERENT!)

- **Write-Write conflicts (lost update)**
  - Or any write over uncommitted data.

Conflict-Serializable Schedules

- **Interchangeability:**
  - If two actions do not conflict, we can interchange them and get an equivalent schedule

- **Two schedules S1 and S2 are conflict equivalent if:**
  - They involve the same set of actions and they order conflicting actions the same way
  - There is a sequence of interchanges of non-conflicting actions that transforms S1 to S2.

- **Schedule S is conflict serializable if S is conflict equivalent to some serial schedule**
  - i.e., if there is a sequence of interchanges of non-conflicting actions that transforms S to a serial schedule
Which Schedules are Conflict Serializable?

T1: R(A)    W(A)
T2: R(A)    W(A), R(B)

T1: R(B)    W(A), W(B)
T2: R(A)    W(A), R(B)

T1: R(A)    R(B)    W(B)
T2:         W(A)    W(A)    R(B)

Precedence Graph

- **Precedence graph**: One node per Xact; edge from $T_i$ to $T_j$ if an action in $T_i$ occurs before an action in $T_j$ and they conflict.
- **Theorem**: Schedule is conflict serializable if and only if its dependency graph is **acyclic**.
- Draw the precedence graph for the previous 3 schedules.
Conflict Serializable ⇒ Serializable

- Every conflict-serializable schedule S is serializable. **Proof:**
  - We can interchange non-conflicting actions to transform S to a serial schedule R.
  - The interchanges produce equivalent schedules
  - Therefore the serial schedule R is equivalent to S.
  - Thus S is serializable.

- Not conversely:
  
  T1:  R(A)    W(A)
  T2:    W(A)
  T3:             W(A)

Is a schedule S Serializable or not?

- How to prove that S is serializable
  - Based on the definition of serializable, we can find a serial schedule that is equivalent to it.
  - We can also show that its precedence graph is acyclic.
    - This proves that it is conflict serializable, therefore serializable.

- How to prove that S is not serializable
  - Enumerate all serial schedules and show that S is not equivalent to any of them - impractical
Locking: Used in Concurrency Control

- Transaction must get a lock – before then can read or update data

- There are two kinds of locks: shared (S) locks and exclusive (X) locks

- To read a record you MUST get an S lock
  To modify or delete a record you MUST get an X lock

- Lock info maintained by a “lock manager”

How Locks Work

- If an object has an S lock, new transactions can get S locks but not X locks.

<table>
<thead>
<tr>
<th>lock you want</th>
<th>--</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>ok</td>
<td>ok</td>
<td>no</td>
</tr>
<tr>
<td>X</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

- If an object has an X lock, no other transaction can get any lock (S or X) on that object.

- If a transaction can’t get a lock, it waits (in a queue).
Strict Two Phase Locking Protocol (2PL)

Strict 2PL is a way of managing locks during a transaction
- T gets (S and X) locks gradually, as needed
- T hold all locks until end of transaction (commit/abort)

![Diagram showing # of locks held by a transaction T over time, with all locks released at the end upon commit or abort.]

Strict 2PL guarantees serializability (it’s been proven!)

<table>
<thead>
<tr>
<th></th>
<th>T1, T2 mixed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>1000</td>
<td>500</td>
</tr>
<tr>
<td>1100</td>
<td></td>
</tr>
<tr>
<td>1166</td>
<td></td>
</tr>
<tr>
<td></td>
<td>530</td>
</tr>
<tr>
<td></td>
<td>430</td>
</tr>
</tbody>
</table>

Illegal values! This is impossible – using Strict 2PL.
Try to enforce 2PL on each schedule

T1: R(A) W(A)
T2: R(A) W(A), R(B)

T1: R(B) W(A), W(B)
T2: R(A) W(A), R(B)

T1: R(A) R(B) W(B)
T2: W(A)
T3: W(A) R(B)

What is deadlock?

- **Deadlocks** can happen whenever you allow a transaction to wait for a lock.

- **Deadlock** is a cycle of transactions, T1, T2, ..., Tn, where each Ti is waiting for its predecessor to release a lock.

  Bob: “I'll wait for you to call and then we'll go eat.”
  Tom (not paying much attention): “Ok, I'll wait for you to call and then we'll go eat.”
  Bob (also not paying much attention): “Alright.”
  DEADLOCK! They'll never eat!
DBMS must also avoid “livelock”

If... a DBMS permits deadlock
and
the DBMS uses deadlock detection, with victim selection
then...

the DBMS must be sure not to select the same victim over and over again. (This is called livelock; it prevents one, unlucky transaction from ever running.)

What should we lock?

- Lock tables – not very much concurrency – lock table is small

- Lock rows – much more concurrency – lock table is much larger

- Lock attribute values – even more concurrency – much larger lock table – difficult to implement

- Lock a set of objects based on a predicate … e.g., lock all students with age > 25 (locks students who appear later – avoids “phantoms”); hard to implement
PHANTOMS!!!

- Booga-booga
- Occurs when different reads of a collection see two different sets of objects
  - New rows show up, despite the current transaction not making any changes
- Here, conflict-serializability does not guarantee serializability

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Isolated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>5</td>
<td>True</td>
</tr>
<tr>
<td>Ted</td>
<td>7</td>
<td>True</td>
</tr>
<tr>
<td>Sally</td>
<td>4</td>
<td>False</td>
</tr>
<tr>
<td>Pratibha</td>
<td>6</td>
<td>True</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Isolated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>5</td>
<td>True</td>
</tr>
<tr>
<td>Ted</td>
<td>7</td>
<td>True</td>
</tr>
<tr>
<td>Sally</td>
<td>4</td>
<td>False</td>
</tr>
<tr>
<td>Pratibha</td>
<td>6</td>
<td>True</td>
</tr>
<tr>
<td>Andrew</td>
<td>5</td>
<td>True</td>
</tr>
</tbody>
</table>

Isolation levels (DBMS’s may differ…)

Developers can choose how much isolation (protection) they want. SQL defines four isolation levels:

- “READ UNCOMMITTED” – allows dirty read, unrepeatable read, and “phantoms”
- “READ COMMITTED” – allows unrepeatable reads and phantoms
- “REPEATABLE READ” – allows phantoms
- “SERIALIZABLE” – full isolation
Isolation levels: How they work

- READ UNCOMMITTED
  - Does not bother with locks when reading
- READ COMMITTED
  - Obtains write locks, releases read locks immediately
- REPEATABLE READ
  - Locks only individual objects
- SERIALIZABLE
  - Locks sets of objects (tables, pages)

Logging and Recovery to provide Durability and Atomicity
Review: The ACID properties

- **Atomicity:** All actions in the transaction happen in their entirety or none of them happen.
- **Consistency:** If each transaction is consistent, and the DB starts in a consistent state, it ends in a consistent state.
- **Isolation:** Execution of one transaction is isolated from that of other transactions.
- **Durability:** If a transaction commits, its effects persist.

Recovery System

Programmers

Concurrency Control System

Recovery System

When do we write DB pages? (using terms in the book)

- **Force** writes to disk immediately?
  Poor response time. But durable.

- **Steal** buffer-pool frames from uncommitted transactions? (Write to DB before commit?)
  - If DBMS doesn’t steal – poor throughput.
  - If DBMS does steal – how can we ensure atomicity?

<table>
<thead>
<tr>
<th></th>
<th>No Steal</th>
<th>Steal</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Force recovery; very slow</td>
<td>Desired; but hard to implement</td>
</tr>
</tbody>
</table>

Some slides taken and adapted from R. Ramakrishnan, with permission.
Consequences of No Force

- Dirty pages may remain in memory after commit.
  - What if we lose memory because of a crash?
  - Updates in the dirty pages are lost. This could kill durability.

- Example:
  - Xact withdrawing and depositing $100 runs and commits
  - The page containing the withdrawal is forced/written to disk.
  - The page containing the deposit is not forced to disk.
  - The page containing the deposit of $100 stays in memory after commit. Crash.
  - Have you lost $100?
  - Would it help if the withdrawal info had been written to a log?
  - We must be able to REDO transactions (reapply them to DB).

Consequences of Steal

- Dirty pages may be written to disk before commit.
  - What if the transaction is aborted?
  - Changes written to disk can’t be reversed. This could kill atomicity.

- Example:
  - Xact deposits $100
  - Page with deposit info is written to disk. Aborts.
  - Have you gained $100?
  - Would it help if the deposit had been logged?
  - We must be able to UNDO the write to P
Motivation for Recovery

Desired Behavior after system restarts:
- T1, T2 & T3 committed (before the crash) and must be durable (effects must be seen).
- T4 & T5 were still running and must be aborted (effects must not be seen).
- T6 was aborted before the crash (effects must not be seen)

Assumptions

- Concurrency control is in effect - *Strict 2PL*
- Updates are happening “in place”. That is, data is overwritten on (or deleted from) the disk.
Simple Model for Recovery: Write-Ahead Logging (WAL)

Always write to the log before writing to the DB. **Force** the log record out to disk immediately.

After each log record is written, write to the DB page (no force). DB page will be written to disk sometime later.
Logging (simple model)

- Put the log on a separate disk.
- Write a log entry for insert, update, delete, begin-trans, commit, abort, and checkpoint.
- Log record contains:
  \(<XID, \text{pageID}, \text{offset}, \text{length}, \text{old data}, \text{new data}>\)

  \(\uparrow\)

  identification of before image after image the record

Transaction Commit (simple model)

- Write commit record to log. (This is immediately written to disk - just like all other log entries - using the simple model.)
Transaction Abort (UNDO) (simple model)

Transaction start 12

\[ \ldots \]

\[ \text{LOG} \]

\[ \text{update} \ldots \text{Before image X} \ldots \text{After image X} \]
\[ \ldots \]

\[ \text{update} \ldots \text{Before image Y} \ldots \text{After image Y} \]
\[ \ldots \]

\[ \text{update} \ldots \text{Before image X} \ldots \text{After image X} \]
\[ \ldots \]

\[ \text{[no commit]} \]

We must reapply BEFORE images in reverse order

How far back do we need to go?

Redo (simple model)

Transaction start 12

\[ \ldots \]

\[ \text{We must reapply AFTER images in forward order} \]

\[ \text{update} \ldots \text{Before image X} \ldots \text{After image X} \]
\[ \ldots \]

\[ \text{update} \ldots \text{Before image Y} \ldots \text{After image Y} \]
\[ \ldots \]

\[ \text{update} \ldots \text{Before image Z} \ldots \text{After image Z} \]
\[ \ldots \]

\[ \text{Commit} \]

\[ \ldots \]

end

How far back do we have to go to start the redo? How many transactions do we need to redo?
Crash Recovery: Big Picture (simple model)

Start from a **checkpoint**.

Three phases. Need to:

- Figure out which transactions committed and which failed since checkpoint (**Analysis**).
- **REDO** actions since the checkpoint for all transactions committed since last checkpoint.
- **UNDO** all actions of failed and incomplete transactions.

---

Crash Recovery: Big Picture (simple model)

- **Analysis**: find transactions that must be **REDO** and **UNDO**
  - (read the log backwards to last checkpoint record, make 2 lists)
- **REDO** transactions
  - (read log forwards from checkpoint, apply “after images” in forward order)
- **UNDO** transactions.
  - (read log backwards, apply “before images” in reverse order; stop at checkpoint if quiesce point was used at the checkpoint)