DBMS Architecture

- Query Parser
- Query Rewriter
- Query Optimizer
- Query Executor

- Lock Manager
- Access Methods
- Buffer Manager
- Log Manager

- Concurrency Control
- Recovery

- Disk Space Manager
- DB
Online Transaction Processing (OLTP)

- **Banking:** Multiple programs can touch the same data
- **Ticketing:** Have you tried to get popular concert tickets online?
- **E-commerce:** Have you experienced online shopping during sales?
- **Voting**
- **Telecommunications**
- **Online gaming**

- A lot of (small) reads and writes on the same data
Outline

- Transaction management overview
- Serializability & recoverability
- Lock-based concurrency control
- Efficient B+tree locking
- Optimistic concurrency control
Concurrent User Programs

- **Concurrent execution of user programs**: good for performance.
  - When task 1 is doing I/O, run task 2 to utilize the CPU.
  - Improve *average response time* (average delay that a user task experiences)
  - Improve *system throughput* (number of user tasks processed in each time unit)
Transactions

- User programs may do many things on the data retrieved.
  - E.g., operations on Bob’s bank account.
  - E.g. transfer of money from account A to account B.
  - E.g., search for a ticket, think about it..., and buy it.

- But the DBMS is only concerned about what data is read from/written to the database.

- A transaction is DBMS’s abstract view of a user program, simply, a sequence of reads and writes of DB objects.
**ACID Properties for Concurrency**

- Many users submit xacts, but each user thinks of his as executing by itself.
  - DMBS *interleaves* reads and writes of xacts for concurrency.

- **Consistency**: each xact starts and ends with a consistent state (i.e., satisfying all integrity constraints).
  - E.g., if an IC states that all accounts must have a positive balance, no transaction can violate this rule.

- **Isolation**: execution of one xact appears isolated from others.
  - Nobody else can see the data in its intermediate state, e.g., account A being debited but B not being credited.
ACID Properties for Recovery

- A transaction might *commit* after completing all its actions, or it could be *aborted* after executing some actions.

- **Atomicity**: either all actions of a xact are performed or none of them is *(all-or-none)*.
  - DBMS *logs* all actions so that it can *undo* the actions of aborted xacts.

- **Durability**: once a user program has been notified of success, its effect will persist despite system failure.
  - DBMS *logs* all actions so that it can *redo* the actions of committed xacts.
Outline

- Transaction management overview
- Serializability & recoverability
- Lock-based concurrency control
- Efficient B+tree locking
- Other locking issues
- Optimistic concurrency control
Example

Consider two transactions:

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

- 1st xact transfers $100 from B’s account to A’s.
- 2nd xact credits both accounts with a 6% interest payment.
- No guarantee that T1 will execute before T2 or vice-versa, if both are submitted together.

However, the net effect must be *equivalent to* these two transactions running *serially* in some order!
Example (Contd.)

- Consider a possible interleaving schedule:

<table>
<thead>
<tr>
<th>Schedule 1</th>
<th>Schedule 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=A+100, B=B-100</td>
<td>A=1.06<em>A, B=1.06</em>B</td>
</tr>
</tbody>
</table>

  T1 -> T2

- What about:

<table>
<thead>
<tr>
<th>Schedule 1</th>
<th>Schedule 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=A+100, B=B-100</td>
<td>A=1.06<em>A, B=1.06</em>B</td>
</tr>
</tbody>
</table>

  A=212 B=106

The DBMS’s view of the second schedule:

<table>
<thead>
<tr>
<th>Schedule 1</th>
<th>Schedule 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A), W(A), R(B), W(B)</td>
<td>R(A), W(A), R(B), W(B)</td>
</tr>
</tbody>
</table>

A=212 B=112

?
Scheduling Transactions

- **Serial schedule:** Schedule that does not interleave the actions of different transactions.

- **Equivalent schedules:** For any database state, the effect of executing the first schedule is identical to the effect of executing the second schedule.

- **Serializable schedule:** A schedule that is equivalent to some serial execution of the transactions.
  - If each transaction preserves consistency, every serializable schedule preserves consistency.
Serializability

- **Serializability theory** concerns the schedules of transactions that are **not (explicitly) aborted**.

- Given a set of such xacts, ideally want to allow **any serializable schedule**.
  - Recognizing any serializable schedule as Xact’s are submitted online is highly complex, if possible.

- Instead, allow only a **subset** of serializable schedules that are easy to detect.
Conflict Serializability

- Two schedules are conflict equivalent if:
  - Involve the same actions of the same transactions.
  - Every pair of potentially conflicting actions is ordered the same way.

- Schedule S is conflict serializable if S is conflict equivalent to some serial schedule.

- Given a set of xacts, conflict serializable schedules are a subset of serializable schedules.
  - There are serializable schedules that can’t be detected using conflict serializability.
Dependency Graph

- **Precedence graph:**
  - One node per Xact;
  - Edge from Xact $Ti$ to Xact $Tj$ if an action of $Ti$ *precedes* and potentially *conflicts with* one of $Tj$ ‘s actions ($RW$, $WR$, $WW$ operations on the same object).

- **Theorem:** Schedule is conflict serializable *if and only if* its precedence graph is acyclic.
Example

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

The schedule is not conflict serializable:
- The cycle in the graph reveals the problem. The output of T1 depends on T2, and vice-versa.
Conflict serializability is *sufficient but not necessary* for serializability.

S1: interleaved schedule

\[
\begin{align*}
T1: & \; R(A) \quad W(A) \\
T2: & \; W(A) \\
T3: & \; W(A)
\end{align*}
\]

S2: serial schedule

\[
\begin{align*}
T1: & \; R(A), W(A) \\
T2: & \; W(A) \\
T3: & \; W(A)
\end{align*}
\]
Recoverability

- **Recoverability theory** concerns schedules that involve aborted transactions.

| T1: R(A), W(A) | Abort |
| T2: R(A), W(A) Commit |

Unrecoverable!

- A schedule S is **recoverable** if each xact commits only after all xacts from which it read have committed.
Recoverability (Contd.)

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

Recoverable, but with cascading aborts.

- S avoids cascading rollback if each xact may read only those values written by committed xacts.
S is *strict* if each xact may read and write only objects previously written by committed xacts.

- No cascading aborts.
- Actions of aborted xacts can be simply undone by restoring the original values of modified objects.
Venn Diagram for Schedules

- All schedules
- Recoverable
  - Avoid cascading aborts
- Strict
- Serializable
  - Conflict serializable
- Serial

Committed Xacts

Also Aborted Xacts
Outline

- Transaction management overview
- Serializability & recoverability
  - Lock-based concurrency control
- Efficient B+tree locking
- Other locking issues
- Optimistic concurrency control
(1) **Locking Protocol: Strict 2PL**

- **Strict Two-Phase Locking (Strict 2PL) Protocol:**
  1. Each Xact must obtain a *S* (*shared*) lock on object before reading, an *X* (*exclusive*) lock on object before writing.
  2. If an Xact holds an *X* lock on an object, no other Xact can get a lock (*S* or *X*) on that object.
  3. All locks held by a transaction are released when the transaction completes.

<table>
<thead>
<tr>
<th>Compatibility</th>
<th>Shared</th>
<th>Exclusive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Exclusive</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

- Diagram: #. locks vs. time

---

25
**Strict 2PL (contd.)**

- **Theorem**: Strict 2PL allows only schedules whose precedence graph is acyclic.
  - Strict 2PL only allows *conflict serializable schedules*!

- Strict 2PL is strict with respect to recoverability.
  - Strict 2PL is *recoverable without anomalies related to aborted transactions*.
  - Hence, it simplifies transaction aborts.
Nonstrict 2PL

- Nonstrict Two-Phase Locking Protocol
  1. Each Xact must obtain a S (shared) lock on object before reading, an X (exclusive) lock on object before writing.
  2. If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.
  3. Duration of locks: A transaction cannot request additional locks once it releases any locks.

![Graph showing the change in the number of locks over time (Growing and Shrinking).]
Nonstrict 2PL (contd.)

- **Theorem:** Nonstrict 2PL ensures acyclicity of precedence graph.
  - Nonstrict 2PL only allows *conflict serializable* schedules.
  - An equivalent serial schedule is given by the order of xacts entering their *shrinking phase*.

- Nonstrict 2PL is recoverable but *not strict*!
  - Involves complex abort processing.
  - But allows xacts to go through more quickly.