This lecture discusses how a DBMS schedules interleaved transactions to avoid the anomalies described in the lecture Transactions I.

**Schedules**

A schedule (of interleaved statements) is called *serializable* if its effect on any consistent database instance is equivalent to running the transactions in *some* serial order.

A schedule is called *recoverable* if a transaction $T_1$ that reads values changed by $T_2$ only commits after $T_2$ commits.

The job of the DBMS is to ensure that the only allowed schedules are *serializable* and *recoverable*.

**Serializable . . .**

This schedule is *serializable* because if both transactions commit as shown, then the effect is the same as running $T_1$ and then $T_2$. 

\[
\begin{align*}
T_1 & \quad \text{Time} \quad T_2 \\
R(A) & \quad W(A) \\
W(A) & \quad R(C) \\
W(C) & \quad R(B) \\
\text{Commit} & \quad W(B) \quad \text{Commit}
\end{align*}
\]
... but not recoverable

This schedule is not recoverable because if $T_1$ aborts, then the earlier statements need to be undone; however $T_2$ has already used the value of $A$ and committed.

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>Time</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R(A)$</td>
<td></td>
<td>$R(A)$</td>
</tr>
<tr>
<td>$W(A)$</td>
<td></td>
<td>$W(A)$</td>
</tr>
<tr>
<td>$R(C)$</td>
<td></td>
<td>$R(B)$</td>
</tr>
<tr>
<td>$W(C)$</td>
<td></td>
<td>$W(B)$</td>
</tr>
<tr>
<td>Abort</td>
<td></td>
<td>Commit</td>
</tr>
</tbody>
</table>

Strict Two-Phase Locking

The most widely used locking protocol is Strict Two-Phase Locking (Strict 2PL) which uses two rules.

- A transaction that wishes to read an object first requests a shared lock on that object, while a transaction that wishes to modify an object first requests an exclusive lock on that object.
- All locks held by a transaction are released when the transaction completes (commits or aborts).

Locking

The main way in which a DBMS ensures that only serializable, recoverable schedules are allowed is through locking protocols. A lock is a flag, or indicator, that can be attached to a database object indicating that it is in use by a transaction; a second transaction wishing to use the same DB object may have to wait until the first transaction has finished it.

A locking protocol is a set of rules that determine what types of lock to use in particular situations.

Altering a schedule

Now consider the non-recoverable schedule from above with Strict 2PL.

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>Time</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R(A)$</td>
<td></td>
<td>$R(A)$</td>
</tr>
<tr>
<td>$W(A)$</td>
<td></td>
<td>$S(A)$</td>
</tr>
<tr>
<td>$R(C)$</td>
<td></td>
<td>$R(A)$</td>
</tr>
<tr>
<td>$W(C)$</td>
<td></td>
<td>$X(A)$</td>
</tr>
<tr>
<td>$R(B)$</td>
<td></td>
<td>$W(A)$</td>
</tr>
<tr>
<td>$W(B)$</td>
<td></td>
<td>$W(B)$</td>
</tr>
<tr>
<td>Abort</td>
<td></td>
<td>Commit</td>
</tr>
<tr>
<td>$S(A)$</td>
<td></td>
<td>blocks</td>
</tr>
</tbody>
</table>
**Blocking**

When transaction $T_2$ requests a shared lock (to read $A$) the DBMS cannot grant that until $T_1$ releases its exclusive lock.

Therefore $T_2$ is *blocked* under $T_1$ completes — either by committing or aborting.

Thus the non-recoverable schedule is *disallowed* under the operation of Strict 2PL.

---

**Deadlock**

A transaction $T_1$ is blocked when it is waiting for another transaction, say $T_2$ to release a lock.

But what happens if $T_2$ is also blocked because it is waiting for $T_1$ to release a lock?

In this situation, the two (or more) transactions are *deadlocked* and without DBMS intervention they would both remain blocked indefinitely.

The DBMS must therefore determine when a transaction is deadlocked and abort it. Detecting deadlock can be done by maintaining an explicit list of blocked transactions and who they are waiting for and then detecting cycles in that list. Alternatively a simpler mechanism is simply to abort any transaction that has been blocked for too long, assuming that it is deadlocked.

---

**Performance**

Because it causes some transactions to be blocked, any locking protocol will cause some degradation in performance.

If there are only a few transactions active, then adding another one will cause the total throughput to increase as interleaving allows it to use CPU idle time. As more transactions are added, some blocking occurs and so the increase in throughput is less. Eventually, each additional transaction conflicts with so many others that the total throughput is reduced. At this point the system is said to be *thrashing*.
Granularity

Locking performance can be substantially affected by the granularity of the locking protocol.

For example, suppose that a transaction is going to modify a few rows in a large table — a coarse-grained locking protocol would lock the entire table, while a fine-grained locking protocol would lock just the individual rows that will be affected.

In MySQL

In order to use transactions in MySQL it is necessary to use a transaction-safe database engine — at the current time (2006) that means using either InnoDB or BDB (Berkeley DB).

The default database engine, MyISAM does not support transactions — in effect it behaves as though each statement were a single-statement transaction.

In particular, with MyISAM it is not possible to guarantee the atomicity of more than a single statement.

InnoDB

The InnoDB storage engine does support transactions.

A transaction can be initiated by a user with the statement

```
START TRANSACTION;
```

All statements after that will be deemed to form part of the same transaction until one of the statements

```
COMMIT;
```

or

```
ROLLBACK;
```

occurs.

Example

Suppose students are forming project groups and being entered into the following database table.

```
CREATE TABLE groups(
    id INT,
    sNum INT,
    UNIQUE(sNum) Engine = InnoDB;
```

The UNIQUE keyword will ensure that no student accidentally ends up allocated to more than one group.
Initial data

Suppose that the first group has been entered:

```sql
SELECT * FROM groups;

+------+------+
<table>
<thead>
<tr>
<th>id</th>
<th>sNum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1537</td>
</tr>
<tr>
<td>1</td>
<td>1433</td>
</tr>
</tbody>
</table>
+------+------+
```

and that the second proposed group comprises students 1010 and 1537; of course 1537 is already in a group and so adding this group should fail.

```
USE a transaction

START TRANSACTION;

INSERT INTO groups VALUES(2,1010);
INSERT INTO groups VALUES(2,1537);

ERROR 1062 (23000): Duplicate entry '1537' for key 1

The first student was correctly added, but the second violates the key constraint — now the entire transaction can be aborted by issuing the ROLLBACK statement.

ROLLBACK;

SELECT * FROM groups;

+------+------+
<table>
<thead>
<tr>
<th>id</th>
<th>sNum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1537</td>
</tr>
<tr>
<td>1</td>
<td>1433</td>
</tr>
</tbody>
</table>
+------+------+
```

Rolling Back

In addition to explicitly issuing a ROLLBACK statement, this will happen automatically if the client becomes disconnected before the transaction is committed. Therefore critical transactions will never get left half-done.

There is also the facility to partially roll-back a transaction to an intermediate “checkpoint” that is declared inside a transaction.

```sql
SAVEPOINT doneOne;
```

If something goes wrong in the next few statements, the user can ROLLBACK TO SAVEPOINT doneOne; rather than rolling back the entire transaction.

Isolation Levels

MySQL permits the user to choose how “isolated” they wish each transaction to be by choosing between

- READ UNCOMMITTED
- READ COMMITTED
- REPEATABLE READ
- SERIALIZABLE

The user can set the isolation level on a per-session or even per-transaction basis, using statements such as:

```sql
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED
```
Isolation levels

The READ UNCOMMITTED and READ COMMITTED isolation levels determine when a transaction will see a value that has been changed by another transaction — either immediately or only after the other transaction has committed.

The default isolation level of REPEATABLE READ guarantees that a transaction will get the same results each time it issues the same SELECT statement. (MySQL also guarantees that REPEATABLE READ prevents phantoms.)

The top level of isolation, SERIALIZABLE is essentially like REPEATABLE READ with minor differences to exactly which statements the guarantee applies to.

### Isolation Level Summary

This table summarizes the anomalies that can or cannot arise at the different isolation levels.

<table>
<thead>
<tr>
<th>Level</th>
<th>Dirty Read</th>
<th>Unrepeatable Read</th>
<th>Phantom</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ UNCOMMITTED</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>No</td>
<td>No</td>
<td>No(^1)</td>
</tr>
<tr>
<td>SERIALIZABLE</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

\(^1\)MySQL-specific