Lecture 15: Optimistic Concurrency Control
Recap
Optimistic Concurrency Control

Recap

Basic T/O

- Txns read and write objects without locks.
- Every object X is tagged with timestamp of the last txn that successfully did read/write:
  - \( W - TS(X) \) – Write timestamp on X
  - \( R - TS(X) \) – Read timestamp on X
- Check timestamps for every operation:
  - If txn tries to access an object \textbf{from the future}, it aborts and restarts.
Partition-based T/O

- Split the database up in disjoint subsets called horizontal partitions (aka shards).
- Use timestamps to order txns for serial execution at each partition.
  - Only check for conflicts between txns that are running in the same partition.
Observation

- If you assume that conflicts between txns are rare and that most txns are short-lived, then forcing txns to wait to acquire locks adds a lot of overhead.
- A better approach is to optimize for the no-conflict case.
Today’s Agenda

- Optimistic Concurrency Control
- Phantoms
- Isolation Levels
Optimistic Concurrency Control
Optimistic Concurrency Control

• The DBMS creates a private workspace for each txn.
  ▶ Any object read is copied into workspace.
  ▶ Modifications are applied to workspace.

• When a txn commits, the DBMS compares workspace write set to see whether it conflicts with other txns.

• If there are no conflicts, the write set is installed into the global database.
OCC Phases

- **Phase 1 – Read:**
  - Track the read/write sets of txns and store their writes in a private workspace.

- **Phase 2 – Validation:**
  - When a txn commits, check whether it conflicts with other txns.

- **Phase 3 – Write:**
  - If validation succeeds, apply private changes to database. Otherwise abort and restart the txn.
OCC – Example

Schedule

- $T_1$
  - BEGIN
  - READ
  - R(A)
  - W(A)
  - VALIDATE
  - WRITE
  - COMMIT

- $T_2$
  - BEGIN
  - READ
  - R(A)
  - VALIDATE
  - WRITE
  - COMMIT

Database

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
OCC – Example
Optimistic Concurrency Control

OCC – Example

Schedule

\[
\begin{array}{c|c}
T_1 & T_2 \\
\hline
\text{BEGIN} & \text{BEGIN} \\
\text{READ} & \text{READ} \\
R(A) & R(A) \\
\text{VALIDATE} & \text{VALIDATE} \\
\text{WRITE} & \text{WRITE} \\
\text{COMMIT} & \text{COMMIT} \\
\end{array}
\]

Database

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

\[
\begin{array}{c|c|c}
T_1 \text{ Workspace} & \\
\hline
\text{Object} & \text{Value} & \text{W-TS} \\
A & 123 & 0 \\
- & - & - \\
\end{array}
\]
OCC – Example
OCC – Example

**Schedule**
- \( T_1 \):
  - BEGIN
  - \text{READ} R(A)
  - W(A)
  - VALIDATE
  - WRITE
  - COMMIT
- \( T_2 \):
  - BEGIN
  - \text{READ} R(A)
  - VALIDATE
  - WRITE

**Database**

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>0</td>
</tr>
</tbody>
</table>

**T_1 Workspace**

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>0</td>
</tr>
</tbody>
</table>

**T_2 Workspace**

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>0</td>
</tr>
</tbody>
</table>

\( TS(T_2) = 1 \)
OCC – Example

Schedule

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>BEGIN</td>
</tr>
<tr>
<td>READ</td>
<td>READ</td>
</tr>
<tr>
<td>R(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>VALIDATE</td>
</tr>
<tr>
<td>VALIDATE</td>
<td>WRITE</td>
</tr>
<tr>
<td>WRITE</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

TS(T₂) = 1

Database

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

T₁ Workspace

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>456</td>
<td>∞</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
OCC – Example

Schedule:
- \( T_1 \): BEGIN
- \( R(A) \), \( W(A) \), VALIDATE, WRITE, COMMIT
- \( T_2 \): BEGIN
- \( R(A) \), \( T_{S(T_2)} = 1 \)

Database:
- Object | Value | W-TS
- A | 456 | 2
- - | - | -

T₁ Workspace:
- Object | Value | W-TS
- A | 456 | ∞
OCC – Validation Phase

- The DBMS needs to guarantee only serializable schedules are permitted.
- \( T_i \) checks other txns for RW and WW conflicts and makes sure that all conflicts go one way (from older txns to younger txns).
OCC – Serial Validation

- Maintain global view of all active txns.
- Record read set and write set while txns are running and write into private workspace.
- Execute **Validation** and **Write** phase inside a protected **critical section**.
OCC – Read Phase

- Track the read/write sets of txns and store their writes in a private workspace.
- The DBMS copies every tuple that the txn accesses from the shared database to its workspace ensure repeatable reads.
OCC – Validation Phase

- Each txn’s timestamp is assigned at the **beginning of the validation phase** (different from 2PL).
- Check the timestamp ordering of the committing txn with all other running txns.
- If $\text{TS}(T_i) < \text{TS}(T_j)$, then one of the following three scenarios must hold...
OCC – Validation Phase

- When the txn invokes COMMIT, the DBMS checks if it conflicts with other txns.
- Two methods for this phase:
  - Backward Validation
  - Forward Validation
OCC – Backward Validation

- Check whether the committing txn intersects its read/write sets with those of any txns that have already committed.
OCC – Backward Validation

- Check whether the committing txn intersects its read/write sets with those of any txns that have already committed.
OCC – Forward Validation

- Check whether the committing txn intersects its read/write sets with any active txns that have not yet committed.
OCC – Forward Validation

- Check whether the committing txn intersects its read/write sets with any active txns that have not yet committed.
OCC – Validation Step 1

- **Scenario 1:**
- $T_i$ completes all three phases before $T_j$ begins.
OCC – Validation Step 2

- **Scenario 2:**
  - $T_i$ completes before $T_j$ starts its **Write** phase, and $T_i$ does not write to any object read by $T_j$.
  - $\text{WriteSet}(T_i) \cap \text{ReadSet}(T_j) = \emptyset$
OCC – Validation Step 2

T₁ has to abort even though T₂ will never write to the database.
OCC – Validation Step 2

**Schedule**

- **T₁**
  - BEGIN
  - READ R(A)
  - W(A)
  - VALIDATE
  - WRITE
  - COMMIT

- **T₂**
  - BEGIN
  - READ R(A)
  - VALIDATE

**Database**

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**T₁ Workspace**

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>456</td>
<td>∞</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**T₂ Workspace**

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Safe to commit T₁ because we know that T₂ will not write.
OCC – Validation Step 3

- **Scenario 3:**
- Ti completes its Read phase before Tj completes its Read phase
- And Ti does not write to any object that is either read or written by Tj:
  - $\text{WriteSet}(Ti) \cap \text{ReadSet}(Tj) = \emptyset$
  - $\text{WriteSet}(Ti) \cap \text{WriteSet}(Tj) = \emptyset$
OCC – Validation Step 3

Schedule:

<table>
<thead>
<tr>
<th></th>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>READ R(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITE W(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VALIDATE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Database:

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>XYZ</td>
<td>0</td>
</tr>
</tbody>
</table>

T₁ Workspace:

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>456</td>
<td>∞</td>
</tr>
</tbody>
</table>

T₂ Workspace:

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>XYZ</td>
<td>0</td>
</tr>
</tbody>
</table>

**Safe to commit T₁ because T₂ sees the DB after T₁ has executed.**

TS(T₁) = 1
OCC – Validation Step 3

**Schedule**

<table>
<thead>
<tr>
<th>Time</th>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BEGIN</td>
<td>BEGIN</td>
</tr>
<tr>
<td></td>
<td>READ</td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td></td>
<td>VALIDATE</td>
<td>VALIDATE</td>
</tr>
<tr>
<td></td>
<td>WRITE</td>
<td>WRITE</td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

**Database**

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>456</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>XYZ</td>
<td>0</td>
</tr>
</tbody>
</table>

**T₁ Workspace**

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>456</td>
<td>∞</td>
</tr>
<tr>
<td>B</td>
<td>XYZ</td>
<td>0</td>
</tr>
</tbody>
</table>

**T₂ Workspace**

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>XYZ</td>
<td>0</td>
</tr>
</tbody>
</table>

**TS(T₁) = 1**
OCC – Validation Step 3

Schedule

T₁

BEGIN
READ
R(A)
W(A)
VALIDATE
WRITE
COMMIT

T₂

BEGIN
READ
R(B)
VALIDATE
WRITE
COMMIT

Database

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>456</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>XYZ</td>
<td>0</td>
</tr>
</tbody>
</table>

T₂ Workspace

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>XYZ</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>456</td>
<td>1</td>
</tr>
</tbody>
</table>
OCC – Observation

• OCC works well when the number of conflicts is low:
  ▶ All txns are read-only (ideal).
  ▶ Txns access disjoint subsets of data.

• If the database is large and the workload is not skewed, then there is a low probability of conflict, so again locking is wasteful.
OCC – Performance Issues

- High overhead for copying data locally.
- Validation/Write phase bottlenecks.
- Aborts are more wasteful than in 2PL because they only occur after a txn has already executed.
Observation

- Recall that so far we have only dealing with transactions that read and update data.
- But now if we have insertions, updates, and deletions, we have new problems. . .
Phantoms
The Phantom Problem

```
CREATE TABLE people (
    id SERIAL,
    name VARCHAR,
    age INT,
    status VARCHAR
);
```
The Phantom Problem

CREATE TABLE people (id SERIAL, name VARCHAR, age INT, status VARCHAR);

BEGIN
SELECT MAX(age) FROM people WHERE status='lit'
BEGIN
INSERT INTO people (age, status) VALUES (96, 'lit')
COMMIT

SELECT MAX(age) FROM people WHERE status='lit'
COMMIT

T_1
T_2

72
96
The Phantom Problem

- How did this happen?
  - Because $T_1$ locked only existing records and not ones under way!
- Conflict serializability on reads and writes of individual items guarantees serializability only if the set of objects is fixed.
Predicate Locking

- Lock records that satisfy a logical predicate:
  - Example: status = 'lit'
- In general, predicate locking has a lot of locking overhead.
- Index locking is a special case of predicate locking that is potentially more efficient.
Index Locking

- If there is a dense index on the status field then the txn can lock index page containing the data with status = 'lit'.
- If there are no records with status = 'lit', the txn must lock the index page where such a data entry would be, if it existed.
Locking without an Index

- If there is no suitable index, then the txn must obtain:
  - A lock on every page in the table to prevent a record’s status = 'lit' from being changed to lit.
  - The lock for the table itself to prevent records with status = 'lit' from being added or deleted.
Repeating Scans

- An alternative is to just re-execute every scan again when the txn commits and check whether it gets the same result.
  - Have to retain the scan set for every range query in a txn.
Weaker Levels of Isolation

• Serializability is useful because it allows programmers to ignore concurrency issues.
• But enforcing it may allow too little concurrency and limit performance.
• We may want to use a weaker level of consistency to improve scalability.
Isolation Levels
Isolation Levels

- Controls the extent that a txn is exposed to the actions of other concurrent txns.
- Provides for greater concurrency at the cost of exposing txns to uncommitted changes:
  - Dirty Reads
  - Unrepeatable Reads
  - Phantom Reads
Isolation Levels

- Isolation (High→Low)
- **SERIALIZABLE**: No phantoms, all reads repeatable, no dirty reads.
- **REPEATABLE READS**: Phantoms may happen.
- **READ COMMITTED**: Phantoms and unrepeatable reads may happen.
- **READ UNCOMMITTED**: All of them may happen.
## Isolation Levels

<table>
<thead>
<tr>
<th>Level</th>
<th>Dirty Read</th>
<th>Unrepeatable Read</th>
<th>Phantom</th>
</tr>
</thead>
<tbody>
<tr>
<td>SERIALIZABLE</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>No</td>
<td>No</td>
<td>Maybe</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>No</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td>READ UNCOMMITTED</td>
<td>Maybe</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
</tbody>
</table>
Isolation Levels

- **SERIALIZABLE**: Obtain all locks first; plus index locks, plus strict 2PL.
- **REPEATABLE READS**: Same as above, but no index locks.
- **READ COMMITTED**: Same as above, but S locks are released immediately.
- **READ UNCOMMITTED**: Same as above, but allows dirty reads (no S locks).
SQL-92 Isolation Levels

- You set a txn’s isolation level before you execute any queries in that txn.
- Not all DBMSs support all isolation levels in all execution scenarios
  - Replicated Environments
- The default depends on implementation...

```sql
SET TRANSACTION Isolation LEVEL <isolation-level>;
BEGIN TRANSACTION ISOLATION LEVEL <isolation-level>;
```
# Isolation Levels (2013)

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Default</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actian Ingres 10.0/10S</td>
<td>SERIALIZABLE</td>
<td>SERIALIZABLE</td>
</tr>
<tr>
<td>Aerospike</td>
<td>READ COMMITTED</td>
<td>READ COMMITTED</td>
</tr>
<tr>
<td>Greenplum 4.1</td>
<td>READ COMMITTED</td>
<td>SERIALIZABLE</td>
</tr>
<tr>
<td>MySQL 5.6</td>
<td>REPEATABLE READS</td>
<td>SERIALIZABLE</td>
</tr>
<tr>
<td>MemSQL 1b</td>
<td>READ COMMITTED</td>
<td>READ COMMITTED</td>
</tr>
<tr>
<td>MS SQL Server 2012</td>
<td>READ COMMITTED</td>
<td>SERIALIZABLE</td>
</tr>
<tr>
<td>Oracle 11g</td>
<td>READ COMMITTED</td>
<td>SNAPSHOT ISOLATION</td>
</tr>
<tr>
<td>Postgres 9.2.2</td>
<td>READ COMMITTED</td>
<td>SERIALIZABLE</td>
</tr>
<tr>
<td>SAP HANA</td>
<td>READ COMMITTED</td>
<td>SERIALIZABLE</td>
</tr>
<tr>
<td>ScaleDB 1.02</td>
<td>READ COMMITTED</td>
<td>READ COMMITTED</td>
</tr>
<tr>
<td>VoltDB</td>
<td>SERIALIZABLE</td>
<td>SERIALIZABLE</td>
</tr>
</tbody>
</table>

- **Source**
SQL-92 Access Modes

- You can provide hints to the DBMS about whether a txn will modify the database during its lifetime.
- Only two possible modes:
  - **READ WRITE** (Default)
  - **READ ONLY**
- Not all DBMSs will optimize execution if you set a txn to in **READ ONLY** mode.

```
SET TRANSACTION <access-mode>;
BEGIN TRANSACTION <access-mode>;
```
Conclusion
Parting Thoughts

- Every concurrency control can be broken down into the basic concepts that I have described in the last two lectures.
  - Two-Phase Locking (2PL): Assumption that collisions are commonplace
  - Timestamp Ordering (T/O): Assumption that collisions are rare.
- Optimistic protocols defer the validation phase to the end of the txn
- I am not showing benchmark results because I don’t want you to get the wrong idea.
Next Class

- Multi-Version Concurrency Control