Last Class

- Serializability
- Two-Phase Locking
- Deadlocks
- Lock Granularities

Today's Class

- Basic Timestamp Ordering
- Optimistic Concurrency Control
- Multi-Version Concurrency Control
- The Phantom Problem
- Weaker Isolation Levels

Concurrency Control Approaches

- **Two-Phase Locking (2PL)**
  - Determine serializability order of conflicting operations at runtime while txns execute.

- **Timestamp Ordering (T/O)**
  - Determine serializability order of txns before they execute.
### Timestamp Allocation

- Each txn $T_i$ is assigned a unique fixed timestamp that is monotonically increasing.
  - Let $TS(T_i)$ be the timestamp allocated to txn $T_i$
  - Different schemes assign timestamps at different times during the txn.
- Multiple implementation strategies:
  - System Clock.
  - Logical Counter.
  - Hybrid.

### T/O Concurrency Control

- Use these timestamps to determine the serializability order.
- If $TS(T_i) < TS(T_j)$, then the DBMS must ensure that the execution schedule is equivalent to a serial schedule where $T_i$ appears before $T_j$.

### 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 “from the future”, it aborts and restarts.

### Basic T/O – Reads

- If $TS(T_i) < W-TS(X)$, this violates timestamp order of $T_i$ w.r.t. writer of $X$.
  - Abort $T_i$ and restart it (with same TS? why?)
- Else:
  - Allow $T_i$ to read $X$.
  - Update $R-TS(X)$ to $\max(R-TS(X), TS(T_i))$
  - Have to make a local copy of $X$ to ensure repeatable reads for $T_i$. 

Faloutsos/Pavlo  CMU SCS 15-415/615  5

Faloutsos/Pavlo  CMU SCS 15-415/615  6

Faloutsos/Pavlo  CMU SCS 15-415/615  7

Faloutsos/Pavlo  CMU SCS 15-415/615  8
**Basic T/O – Writes**

- If $\text{TS}(T_i) < R-\text{TS}(X)$ or $\text{TS}(T_i) < W-\text{TS}(X)$
  - Abort and restart $T_i$.
- Else:
  - Allow $T_i$ to write $X$ and update $W-\text{TS}(X)$
  - Also have to make a local copy of $X$ to ensure repeatable reads for $T_i$.

---

**Basic T/O – Example #1**

**Schedule**

<table>
<thead>
<tr>
<th>Time</th>
<th>Object</th>
<th>R-TS</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>B</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Database**

<table>
<thead>
<tr>
<th>Time</th>
<th>Object</th>
<th>R-TS</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Basic T/O – Example #1

Database

Object | R-TS | W-TS
--- | --- | ---
A | 2 | 2
B | 2 | 2

No violations so both txns are safe to commit.

Basic T/O – Example #2

Schedule

Object | R-TS | W-TS
--- | --- | ---
A | 0 | 0
B | - | -

Basic T/O – Example #2

Schedule

Object | R-TS | W-TS
--- | --- | ---
A | 1 | 2
B | - | -

Basic T/O – Example #2

Schedule

<table>
<thead>
<tr>
<th>Object</th>
<th>R-TS</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Database

Violation: TS(T1) < W-TS(A)

T1 cannot overwrite update by T2, so it has to abort+restart.

Basic T/O – Thomas Write Rule

- If TS(Ti) < R-TS(X):
  - Abort and restart Ti.
- If TS(Ti) < W-TS(X):
  - Thomas Write Rule: Ignore the write and allow the txn to continue.
  - This violates timestamp order of Ti
- Else:
  - Allow Ti to write X and update W-TS(X)
Basic T/O – Thomas Write Rule

Schedule

\[ \begin{array}{c|c|c|c|}
\text{T1} & \text{T2} \\
\hline
\text{BEGIN} & \text{BEGIN} \\
\text{R(A)} & \text{W(A)} \\
\text{COMMIT} & \text{COMMIT} \\
\end{array} \]

Database

\[ \begin{array}{c|c|c|}
\text{Object} & \text{R-TS} & \text{W-TS} \\
\hline
A & 1 & 2 \\
- & - & - \\
- & - & - \\
\end{array} \]

1. Ignore the write and allow T1 to commit.
2. We do not update \( W-TS(A) \).

Basic T/O

• Ensures conflict serializability if you don’t use the Thomas Write Rule.
• No deadlocks because no txn ever waits.
• Possibility of starvation for long txns if short txns keep causing conflicts.
• Permits schedules that are not recoverable.

Recoverable Schedules

• Transactions commit only after all transactions whose changes they read, commit.
Recoverability

T2 is allowed to read the writes of T1.

This is not recoverable because we can’t restart T2.

T1 aborts after T2 has committed.

Today's Class

- Basic Timestamp Ordering
- Optimistic Concurrency Control
- Multi-Version Concurrency Control
- The Phantom Problem
- Weaker Isolation Levels

Optimistic Concurrency Control

- Assumption: Conflicts are rare
- Forcing txns to wait to acquire locks adds a lot of overhead.
- Optimize for the no-conflict case.

OCC Phases

- **Read:** Track the read/write sets of txns and store their writes in a private workspace.
- **Validation:** When a txn commits, check whether it conflicts with other txns.
- **Write:** If validation succeeds, apply private changes to database. Otherwise abort and restart the txn.
### Schedule

#### T1
- BEGIN
- READ R(A)
- VALIDATE
- WRITE
- COMMIT

#### T2
- BEGIN
- READ R(A)
- 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>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>123</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>
</tbody>
</table>

\[\text{TS(T2)} = 1\]
OCC – Example

Schedule

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

T2
BEGIN
READ R(A)
VALIDATE
WRITE
COMMIT

Database

Object | Value | W-TS
--------|-------|-------
A       | 123   | 0     

T1 Workspace

Object | Value | W-TS
-------|-------|-------
A       | 456   | ∞     

OCC – Example

Schedule

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

T2
BEGIN
READ R(A)
VALIDATE
WRITE
COMMIT

Database

Object | Value | W-TS
--------|-------|-------
A       | 123   | 0     

T1 Workspace

Object | Value | W-TS
-------|-------|-------
A       | 456   | ∞     

OCC – Validation Phase

• Need to guarantee only serializable schedules are permitted.
• At validation, 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 – Validation Phase

- Each txn’s timestamp is assigned at the beginning of the validation phase.
- 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 conditions must hold…

OCC – Validation #1

- $T_i$ completes all three phases before $T_j$ begins.

OCC – Validation #1

```
BEGIN
READ
VALIDATE
WRITE
COMMIT
T1
```
```
BEGIN
READ
VALIDATE
WRITE
COMMIT
T2
```
**OCC – Validation #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 #2**

### Schedule

- $T_1$
  - BEGIN
  - READ
    - $R(A)$
  - WRITE
    - $W(A)$
  - VALIDATE
  - 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>0</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

$T_1$ has to abort even though $T_2$ will never write to the database.

**OCC – Validation #3**

- $T_i$ completes its **Read** phase before $T_j$ completes its **Read** phase
- And $T_i$ does not write to any object that is either read or written by $T_j$:
  - $\text{WriteSet}(T_i) \cap \text{ReadSet}(T_j) = \emptyset$
  - $\text{WriteSet}(T_i) \cap \text{WriteSet}(T_j) = \emptyset$

### Schedule

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

- $T_2$
  - BEGIN
  - READ
    - $R(A)$
  - WRITE
    - $A$
  - 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>0</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Safe to commit $T_1$ because we know that $T_2$ will not write.
**OCC – Validation #3**

**Schedule**

- **T1**
  - BEGIN
  - READ
  - R(A)
  - W(A)
  - VALIDATE
  - R(B)
  - 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>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>
<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>A</td>
<td>456</td>
<td>∞</td>
</tr>
<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 – Observations**

- **Q**: When does OCC work well?
  - **A**: When # 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 because they only occur after a txn has already executed.
- Suffers from timestamp allocation bottleneck.

Today's Class

- Basic Timestamp Ordering
- Optimistic Concurrency Control
- Multi-Version Concurrency Control
- The Phantom Problem
- Weaker Isolation Levels

Multi-Version Concurrency Control

- Writes create new versions of objects instead of in-place updates:
  - Each successful write results in the creation of a new version of the data item written.
- Use write timestamps to label versions.
  - Let $X_k$ denote the version of $X$ where for a given txn $T_i$: $W-TS(X_k) \leq TS(T_i)$

MVCC – Reads

- Any read operation sees the latest version of an object from right before that txn started.
- Every read request can be satisfied without blocking the txn.
- If $TS(T_i) > R-TS(X_k)$:
  - Set $R-TS(X_k) = TS(T_i)$
MVCC – Writes

- If $TS(T_i) < R-TS(X_k)$:
  - Abort and restart $T_i$.
- If $TS(T_i) = W-TS(X_k)$:
  - Overwrite the contents of $X_k$.
- Else:
  - Create a new version of $X_{k+1}$ and set its write timestamp to $TS(T_i)$.

**MVCC – Example #1**

```
TIME
BEGIN R(A) W(A)
R(A)
COMMIT
```

Schedule

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>R-TS</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₀</td>
<td>123</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Database

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₀</td>
<td>123</td>
</tr>
<tr>
<td>A₁</td>
<td>456</td>
</tr>
</tbody>
</table>

```
TIME
BEGIN R(A) W(A)
R(A)
COMMIT
```
MVCC – Example #1

T1 reads version A1 that it wrote earlier.

Schedule

Database

Object | Value | R-TS | W-TS
---|---|---|---
A₀ | 123 | 1 | 0
A₁ | 456 | 2 | 1
A₂ | 789 | 2 | 2

TS(T1)=1  
TS(T2)=2

T1 T2
**MVCC – Example #2**

**Schedule**

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

**Database**

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>R-TS</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>123</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

T1 is aborted because T2 “moved” time forward.

Violation: \( TS(T1) < R-TS(A_0) \)

---

**MVCC**

- Can still incur cascading aborts because a txn sees uncommitted versions from txns that started before it did.
- Old versions of tuples accumulate.
- The DBMS needs a way to remove old versions to reclaim storage space.

---

**MVCC Implementations**

- Store versions directly in main tables:
  - Postgres, Firebird/Interbase
- Store versions in separate temp tables:
  - MSFT SQL Server
- Only store a single master version:
  - Oracle, MySQL
Garbage Collection – Postgres

- Never overwrites older versions.
- New tuples are appended to table.
- Deleted tuples are marked with a tombstone and then left in place.
- Separate background threads (VACUUM) has to scan tables to find tuples to remove.

Garbage Collection – MySQL

- Only one “master” version for each tuple.
- Information about older versions are put in temp rollback segment and then pruned over time with a single thread (PURGE).
- Deleted tuples are left in place and the space is reused.

MVCC – Performance Issues

- High abort overhead cost.
- Garbage collection overhead.
- Requires stalls to ensure recoverability.
- Secondary index updates.

Today's Class

- Basic Timestamp Ordering
- Optimistic Concurrency Control
- Multi-Version Concurrency Control
- The Phantom Problem
- Weaker Isolation Levels
Dynamic Databases

• 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...

The Phantom Problem

How did this happen?

• Because T1 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.
• Solution?
Predicate Locking

• Lock records that satisfy a logical predicate:
  – Example: rating=1.
• 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 rating field then the txn can lock index page containing the data with rating=1.
• If there are no records with rating=1, 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 rating from being changed to 1.
  – The lock for the table itself to prevent records with rating=1 from being added or deleted.

Today's Class

• Basic Timestamp Ordering
• Optimistic Concurrency Control
• Multi-Version Concurrency Control
• The Phantom Problem
• Weaker Isolation Levels
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

- 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

- **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.

<table>
<thead>
<tr>
<th>Isolation 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

```
SET TRANSACTION ISOLATION LEVEL <isolation-level>;
```

- Not all DBMS support all isolation levels in all execution scenarios (e.g., replication).
- Default: Depends...

Access Modes

- You can also provide hints to the DBMS about whether a txn will modify the database.
- Only two possible modes:
  - **READ WRITE**
  - **READ ONLY**
### SQL-92 Access Modes

**SQL-92**

```
SET TRANSACTION <access-mode>;
```

**Postgres + MySQL 5.6**

```
START TRANSACTION <access-mode>;
```

- Default: **READ WRITE**
- Not all DBMSs will optimize execution if you set a txn to in **READ ONLY** mode.

### Which CC Scheme is Best?

- Like many things in life, it depends…
  - How skewed is the workload?
  - Are the txns short or long?
  - Is the workload mostly read-only?

### Real Systems

<table>
<thead>
<tr>
<th>Scheme Released</th>
<th>Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ingres</td>
<td>Strict 2PL</td>
</tr>
<tr>
<td>1975</td>
<td></td>
</tr>
<tr>
<td>Informix</td>
<td>Strict 2PL</td>
</tr>
<tr>
<td>1980</td>
<td></td>
</tr>
<tr>
<td>IBM DB2</td>
<td>Strict 2PL</td>
</tr>
<tr>
<td>1983</td>
<td></td>
</tr>
<tr>
<td>Oracle</td>
<td>MVCC</td>
</tr>
<tr>
<td>1984*</td>
<td></td>
</tr>
<tr>
<td>Postgres</td>
<td>MVCC</td>
</tr>
<tr>
<td>1985</td>
<td></td>
</tr>
<tr>
<td>MS SQL Server</td>
<td>Strict 2PL or MVCC</td>
</tr>
<tr>
<td>1992*</td>
<td></td>
</tr>
<tr>
<td>MySQL (InnoDB)</td>
<td>MVCC+2PL</td>
</tr>
<tr>
<td>2001</td>
<td></td>
</tr>
<tr>
<td>Aerospike</td>
<td>OCC</td>
</tr>
<tr>
<td>2009</td>
<td></td>
</tr>
<tr>
<td>SAP HANA</td>
<td>MVCC</td>
</tr>
<tr>
<td>2010</td>
<td></td>
</tr>
<tr>
<td>VoltDB</td>
<td>Partition T/O</td>
</tr>
<tr>
<td>2010</td>
<td></td>
</tr>
<tr>
<td>MemSQL</td>
<td>MVCC</td>
</tr>
<tr>
<td>2011</td>
<td></td>
</tr>
<tr>
<td>MS Hekaton</td>
<td>MVCC+OCC</td>
</tr>
<tr>
<td>2013</td>
<td></td>
</tr>
</tbody>
</table>

### Summary

- Concurrency control is hard.