Last Class

- Lock Granularities
- Locking in B+Trees
- The Phantom Problem
- Transaction 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.
Today's Class

- Basic Timestamp Ordering
- Optimistic Concurrency Control
- Multi-Version Concurrency Control
- Multi-Version+2PL
- Partition-based T/O
- Performance Comparisons

Timestamp Allocation

- Each txn Ti is assigned a unique fixed timestamp that is monotonically increasing.
  - Let $TS(T_i)$ be the timestamp allocated to txn Ti
  - 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 Ti appears before Tj.
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(Ti) < W-TS(X), this violates timestamp order of Ti w.r.t. writer of X.
  – Abort Ti and restart it (with same TS? why?)
• Else
  – Allow Ti to read X.
  – Update R-TS(X) to max(R-TS(X), TS(Ti))
  – Have to make a local copy of X to ensure repeatable reads for Ti.

Basic T/O – Writes

• If TS(Ti) < R-TS(X) or TS(Ti) < W-TS(X)
  – Abort and restart Ti.
• Else
  – Allow Ti to write X and update W-TS(X)
  – Also have to make a local copy of X to ensure repeatable reads for Ti.
Basic T/O – Example #1

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

Violation:
\[ \text{TS}(T1) < \text{W-TS}(A) \]

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

Basic T/O – Thomas Write Rule

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

Schedule
- T1
  - BEGIN R(A)
  - BEGIN W(A)
  - COMMIT
- T2
  - BEGIN W(A)
  - COMMIT

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

Ignore the write and allow T1 to commit.

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.

Recoverability

Schedule
- T1
  - BEGIN W(A)
  - COMMIT
- T2
  - BEGIN R(A)
  - W(B)
  - COMMIT

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.
Basic T/O – Performance Issues

- High overhead from copying data to txn’s workspace and from updating timestamps.
- Long running txns can get starved.
- Suffers from timestamp bottleneck.

Today’s Class

- Basic Timestamp Ordering
- Optimistic Concurrency Control
- Multi-Version Concurrency Control
- Multi-Version+2PL
- Partition-based T/O
- Performance Comparisons

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 its read set overlaps with the write set of any concurrent txns.
- **Write**: If validation succeeds, apply private changes to database. Otherwise abort and restart the txn.

OCC – Example

![Diagram showing transaction schedules and database states](image)

- **TS**: Transaction Start Time
- **W**: Write
- **R**: Read

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 A</td>
<td>READ A</td>
</tr>
<tr>
<td></td>
<td>TS(T1)=1</td>
<td>TS(T2)=1</td>
</tr>
<tr>
<td></td>
<td>VALIDATE</td>
<td>VALIDATE</td>
</tr>
<tr>
<td></td>
<td>WRITE A</td>
<td>WRITE A</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>123</td>
<td>-</td>
</tr>
<tr>
<td>A</td>
<td>456</td>
<td>∞</td>
</tr>
</tbody>
</table>

OCC – Validation Phase

- Need to guarantee only serializable schedules are permitted.
- At validation, Ti checks other txns for RW and WW conflicts and makes sure that all conflicts go one way (from older txns to younger txns).
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 $TS(T_i) < TS(T_j)$, then one of the following three conditions must hold…

OCC – Validation #1

- $T_i$ completes all three phases before $T_j$ begins.
OCC – Validation #2

• Ti completes before Tj starts its Write phase, and Ti does not write to any object read by Tj.
  – WriteSet(Ti) ∩ ReadSet(Tj) = Ø

T1 has to abort even though T2 will never write to the database.

Safe to commit T1 because we know that T2 will not write.
OCC – Validation #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:
  – WriteSet(Ti) \cap ReadSet(Tj) = \emptyset
  – WriteSet(Ti) \cap WriteSet(Tj) = \emptyset

OCC – Validation #3

Schedule

Database

Object | Value | W-TS
---|---|---
A | 456 | 1
B | XYZ | 0

T1 Workspace

Object | Value | W-TS
---|---|---
A | 456 | 1
B | 456 | ∞
A | 456 | 1

T2 Workspace

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 – 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
• Multi-Version+2PL
• Partition-based T/O
• Performance Comparisons
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 \( \text{txn} \) \( Ti \):
    \[ \text{W-} \text{TS}(X_k) \leq \text{TS}(Ti) \]

MVCC – Reads

• Any read operation see the latest version of an object from right before thattxn started.
• Every read request can be satisfied without blocking the txn.
• If \( \text{TS}(Ti) > \text{R-} \text{TS}(X_k) \):
  – Set \( \text{R-} \text{TS}(X_k) = \text{TS}(Ti) \)

MVCC – Writes

• If \( \text{TS}(Ti) < \text{R-} \text{TS}(X_k) \):
  – Abort and restart \( Ti \).
• If \( \text{TS}(Ti) = \text{W-} \text{TS}(X_k) \):
  – Overwrite the contents of \( X_k \).
• Else
  – Create a new version of \( X_{k+1} \) and set its write timestamp to \( \text{TS}(Ti) \).
VMCC – Example #1

T1 reads version A_1 that it wrote earlier.

T1 is aborted because T2 “moved” time forward.

Violation: \( TS(T1) < R-\text{TS}(A_0) \)

- 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.
Garbage Collection – PostgreSQL

- 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.
- Older versions are put into a temporary 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.
- Suffers from timestamp allocation bottleneck.
- Garbage collection overhead.
- Requires stalls to ensure recoverability.
Today's Class

- Basic Timestamp Ordering
- Optimistic Concurrency Control
- Multi-Version Concurrency Control
- Multi-Version+2PL
- Partition-based T/O
- Performance Comparisons

MVCC+2PL

- Combine the advantages of MVCC and 2PL together in a single scheme.
- Use different concurrency control scheme for read-only txns than for update txns.

MVCC+2PL – Reads

- Use MVCC for read-only txns so that they never block on a writer
- Read-only txns are assigned a timestamp when they enter the system.
- Any read operations see the latest version of an object from right before that txn started.
MVCC+2PL – Writes

- Use strict 2PL to schedule the operations of update txns:
  - Read-only txns are essentially ignored.
- Txns never overwrite objects:
  - Create a new copy for each write and set its timestamp to $\infty$.
  - Set the correct timestamp when txn commits.
  - Only one txn can commit at a time.

MVCC+2PL – Performance Issues

- All the lock contention of 2PL.
- Suffers from timestamp allocation bottleneck.

Today's Class

- Basic Timestamp Ordering
- Optimistic Concurrency Control
- Multi-Version Concurrency Control
- Multi-Version+2PL
- Partition-based T/O
- Performance Comparisons
Observation

- When a txn commits, all previous T/O schemes check to see whether there is a conflict with concurrent txns.
- This requires locks/latches/mutexes.
- If you have a lot of concurrent txns, then this is slow even if the conflict rate is low.

Partition-based T/O

- Split the database up in disjoint subsets called partitions (aka shards).
- Only check for conflicts between txns that are running in the same partition.

Database Partitioning

Schema

```
WAREHOUSE
   └─ DISTRICT
       └─ CUSTOMER
          └─ ORDERS
             └─ ORDER_ITEM
                └─ ITEM
```

Schema Tree

```
WAREHOUSE
   └─ DISTRICT
       └─ CUSTOMER
          └─ ORDERS
             └─ ORDER_ITEM
                └─ ITEM
```

Replicated
Database Partitioning

Schema Tree

Partitions

Partition-based T/O

• Txns are assigned timestamps based on when they arrive at the DBMS.
• Partitions are protected by a single lock:
  – Eachtxn is queued at the partitions it needs.
  – The txn acquires a partition’s lock if it has the lowest timestamp in that partition’s queue.
  – The txn starts when it has all of the locks for all the partitions that it will read/write.

Partition-based T/O – Reads

• Do not need to maintain multiple versions.
• Txns can read anything that they want at the partitions that they have locked.
• If a txn tries to access a partition that it does not have the lock, it is aborted + restarted.
Partition-based T/O – Writes

- All updates occur in place.
  - Maintain a separate in-memory buffer to undo changes if the txn aborts.
- If a txn tries to access a partition that it does not have the lock, it is aborted and restarted.

Partition-based T/O – Performance Issues

- Partition-based T/O protocol is very fast if:
  - The DBMS knows what partitions the txn needs before it starts.
  - Most (if not all) txns only need to access a single partition.
- Multi-partition txns cause partitions to be idle while txn executes.

Today's Class

- Basic Timestamp Ordering
- Optimistic Concurrency Control
- Multi-Version Concurrency Control
- Multi-Version+2PL
- Partition-based T/O
- Performance Comparisons
Performance Comparison

- Different schemes make different trade-offs.
- Measure how well each scheme scales on future many-core CPUs.
  - Ignore indexing and logging issues (for now).

Joint work with Xiangyu Yu, George Bezerra, Mike Stonebraker, and Srinik Devadas.

Graphite CPU Simulator

- Simulates a single CPU with 1024 cores.
  - Runs on a 22-node cluster.
  - Average slowdown: 10,000x
- Custom, lightweight DBMS that supports pluggable concurrency control coordinator.

Tested CC Schemes

<table>
<thead>
<tr>
<th>T/O Schemes</th>
<th>2PL Schemes</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL_DETECT</td>
<td>2PL with Deadlock Detection</td>
</tr>
<tr>
<td>NO_WAIT</td>
<td>2PL with Non-waiting Deadlock Prevention</td>
</tr>
<tr>
<td>WAIT_DIE</td>
<td>2PL with Wait-Die Deadlock Prevention</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>Basic T/O</td>
</tr>
<tr>
<td>OCC</td>
<td>Optimistic Concurrency Control</td>
</tr>
<tr>
<td>MVCC</td>
<td>Multi-Version Concurrency Control</td>
</tr>
<tr>
<td>H-STORE</td>
<td>Partition-based T/O</td>
</tr>
</tbody>
</table>
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?
CC Schemes

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Released</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL_DETECT</td>
<td>Scales under low-contention. Suffers from lock thrashing and deadlocks.</td>
</tr>
<tr>
<td>NO_WAIT</td>
<td>Has no centralized point of contention. Highly scalable. Very high abort rates.</td>
</tr>
<tr>
<td>WAIT_DIE</td>
<td>Suffers from lock thrashing and timestamp allocation bottleneck. No deadlocks.</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>High overhead from writing data and timestamp bottleneck. Non-blocking writes.</td>
</tr>
<tr>
<td>OCC</td>
<td>Performs well for read-only workloads. Non-blocking reads and writes. Timestamp bottleneck.</td>
</tr>
<tr>
<td>MVCC</td>
<td>High overhead for copying data locally. High abort cost. Suffers from timestamp bottleneck.</td>
</tr>
<tr>
<td>H-STORE</td>
<td>The best algorithm for partitioned workloads. Suffers from timestamp bottleneck.</td>
</tr>
</tbody>
</table>

2PL Schemes

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Released</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ingres</td>
<td>Strict 2PL</td>
</tr>
<tr>
<td>Informix</td>
<td>Strict 2PL</td>
</tr>
<tr>
<td>IBM DB2</td>
<td>Strict 2PL</td>
</tr>
<tr>
<td>Oracle</td>
<td>MVCC</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>MVCC</td>
</tr>
<tr>
<td>MS SQL Server</td>
<td>Strict 2PL</td>
</tr>
<tr>
<td>MySQL (InnoDB)</td>
<td>MVCC+2PL</td>
</tr>
<tr>
<td>Aerospike</td>
<td>OCC</td>
</tr>
<tr>
<td>SAP HANA</td>
<td>MVCC</td>
</tr>
<tr>
<td>VoltDB</td>
<td>Partition T/O</td>
</tr>
<tr>
<td>MemSQL</td>
<td>MVCC</td>
</tr>
<tr>
<td>MS Hekaton</td>
<td>MVCC+OCC</td>
</tr>
</tbody>
</table>

Real Systems

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Released</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ingres</td>
<td>1975</td>
</tr>
<tr>
<td>Informix</td>
<td>1980</td>
</tr>
<tr>
<td>IBM DB2</td>
<td>1983</td>
</tr>
<tr>
<td>Oracle</td>
<td>1984</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>1985</td>
</tr>
<tr>
<td>MS SQL Server</td>
<td>1992</td>
</tr>
<tr>
<td>MySQL (InnoDB)</td>
<td>2001</td>
</tr>
<tr>
<td>Aerospike</td>
<td>2009</td>
</tr>
<tr>
<td>SAP HANA</td>
<td>2010</td>
</tr>
<tr>
<td>VoltDB</td>
<td>2010</td>
</tr>
<tr>
<td>MemSQL</td>
<td>2011</td>
</tr>
<tr>
<td>MS Hekaton</td>
<td>2013</td>
</tr>
</tbody>
</table>

Summary

- Concurrency control is hard.