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
- Partition-based T/O

- The Phantom Problem
- Weaker Isolation Levels

Timestamp Allocation

- Each txn Ti is assigned a unique fixed timestamp that is monotonically increasing.
  - Let $\text{TS}(Ti)$ 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 $TS(Ti) < TS(Tj)$, 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

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

**Database**

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

No violations so both txns are safe to commit.

Basic T/O – Example #2

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

**Violations:**

- **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)**

We do not update **W-TS(A)**

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.

Recoverable Schedules

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

Recoverability

Schedule

<table>
<thead>
<tr>
<th>TIME</th>
<th>( T1 )</th>
<th>( T2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \begin{align*} &amp; {\text{BEGIN}} \cr &amp; {\text{W(A)}} \cr &amp; {\text{!}} \end{align*} )</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \begin{align*} &amp; {\text{BEGIN}} \cr &amp; {\text{R(A)}} \cr &amp; {\text{W(B)}} \cr &amp; {\text{COMMIT}} \end{align*} )</td>
<td></td>
</tr>
</tbody>
</table>

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
- Partition-based T/O

- 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.

OCC – Example

Schedule

Database

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Database</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td></td>
</tr>
<tr>
<td>BEGIN</td>
<td>A</td>
</tr>
<tr>
<td>READ (A)</td>
<td>Value</td>
</tr>
<tr>
<td>WRITE (A)</td>
<td>W-TS</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td></td>
</tr>
<tr>
<td>BEGIN</td>
<td>A</td>
</tr>
<tr>
<td>READ (A)</td>
<td>Value</td>
</tr>
<tr>
<td>WRITE (A)</td>
<td>W-TS</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Database</th>
<th>T1 Workspace</th>
<th>T2 Workspace</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Object</td>
<td>Value</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>456</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Database</th>
<th>T1 Workspace</th>
<th>T2 Workspace</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Object</td>
<td>Value</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>123</td>
</tr>
</tbody>
</table>
• 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).

• 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.

• 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 $TS(Ti) < TS(Tj)$, then one of the following three conditions must hold…

• Ti completes all three phases before Tj begins.
OCC – Validation #1

BEGIN
READ
VALIDATE
WRITE
COMMIT

T1 T2

BEGIN
READ
VALIDATE
WRITE
COMMIT

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

OCC – Validation #2

Schedule

<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>

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>456</td>
<td>∞</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

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

OCC – Validation #2

Schedule

<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>

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>456</td>
<td>∞</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 #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:
  - $\text{WriteSet}(Ti) \cap \text{ReadSet}(Tj) = \emptyset$
  - $\text{WriteSet}(Ti) \cap \text{WriteSet}(Tj) = \emptyset$

Faloutsos/Pavlo  CMU SCS 15-415/615  30

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.

Faloutsos/Pavlo  CMU SCS 15-415/615  33

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.
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 $T_i$: $W-\text{TS}(X_k) \leq \text{TS}(T_i)$

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 $\text{TS}(T_i) > R-\text{TS}(X_k)$:
  - Set $R-\text{TS}(X_k) = \text{TS}(T_i)$

MVCC – Writes

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

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

BEGIN R(A)
W(A)
COMMIT

BEGIN R(A)
W(A)
COMMIT

Database

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

T1 reads version A1 that it wrote earlier.

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 – Example #2

Schedule

T1 T2

BEGIN
R(A)
COMMIT

BEGIN
R(A)
COMMIT

Database

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

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

Violation:
TS(T1) < R-TS(A0)

T1 is aborted because T2 “moved” time forward.

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.
- Suffers from timestamp allocation bottleneck.
- Garbage collection overhead.
- Requires stalls to ensure recoverability.

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
• Partition-based T/O
• The Phantom Problem
• Weaker Isolation Levels
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       ITEM
   ↓                ↓
DISTRICT        STOCK
   ↓                ↓
CUSTOMER        ORDER_ITEM
   ↓                ↓
ORDERS          ORDER_ITEM
```

Schema Tree

```
WAREHOUSE
   ↓
DISTRICT
   ↓
CUSTOMER
   ↓
ORDERS
   ↓
ORDER_ITEM
```

Replicated

```
ITEM
```

Database Partitioning

Schema Tree

```
WAREHOUSE
   ↓
DISTRICT
   ↓
CUSTOMER
   ↓
ORDERS
   ↓
ORDER_ITEM
```

Replicated

```
ITEM
```

Partitions

```
P1
   ↓
P2
   ↓
P3
   ↓
P4
   ↓
P5
```

```
P1
   ↓
P2
   ↓
P3
   ↓
P4
   ↓
P5
```

```
P1
   ↓
P2
   ↓
P3
   ↓
P4
   ↓
P5
```

```
P1
   ↓
P2
   ↓
P3
   ↓
P4
   ↓
P5
```

```
P1
   ↓
P2
   ↓
P3
   ↓
P4
   ↓
P5
```

```
P1
   ↓
P2
   ↓
P3
   ↓
P4
   ↓
P5
```

```
P1
   ↓
P2
   ↓
P3
   ↓
P4
   ↓
P5
```
Partition-based T/O

- Txns are assigned timestamps based on when they arrive at the DBMS.
- Partitions are protected by a single lock:
  - Each txn 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 + 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 causes partitions to be idle while txn executes.
Today's Class

- Basic Timestamp Ordering
- Optimistic Concurrency Control
- Multi-Version Concurrency Control
- Partition-based T/O
- 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

Schedule

T1

BEGIN
SELECT MAX(age)
FROM sailors
WHERE rating = 1

72

SELECT MAX(age)
FROM sailors
WHERE rating = 1

96

COMMIT

T2

BEGIN
INSERT INTO sailors
(age = 96, rating = 1)

COMMIT

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: \texttt{rating=1}.
- In general, predicate locking has a lot of locking overhead.
- \textbf{Index locking} is a special case of predicate locking that is potentially more efficient.

Index Locking

- If there is a dense index on the \texttt{rating} field then the txn can lock index page containing the data with \texttt{rating=1}.
- If there are no records with \texttt{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 \texttt{rating} from being changed to 1.
  - The lock for the table itself to prevent records with \texttt{rating=1} from being added or deleted.

Today's Class

- Basic Timestamp Ordering
- Optimistic Concurrency Control
- Multi-Version Concurrency Control
- Partition-based T/O
- The Phantom Problem
- Weaker Isolation Levels
Weaker Levels of Consistency

• 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

<table>
<thead>
<tr>
<th></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

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

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

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**

### Default Maximum

<table>
<thead>
<tr>
<th>Database</th>
<th>Default</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actian Ingres 10.0/10S</td>
<td>SERIALIZEABLE</td>
<td>SERIALIZEABLE</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>SERIALIZEABLE</td>
</tr>
<tr>
<td>MySQL 5.6</td>
<td>REPEATABLE READS</td>
<td>SERIALIZEABLE</td>
</tr>
<tr>
<td>MemSQL 1.0b</td>
<td>READ COMMITTED</td>
<td>READ COMMITTED</td>
</tr>
<tr>
<td>MS SQL Server 2012</td>
<td>READ COMMITTED</td>
<td>SERIALIZEABLE</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>SERIALIZEABLE</td>
</tr>
<tr>
<td>SAP HANA</td>
<td>READ COMMITTED</td>
<td>SERIALIZEABLE</td>
</tr>
<tr>
<td>ScaleDB 1.02</td>
<td>READ COMMITTED</td>
<td>READ COMMITTED</td>
</tr>
<tr>
<td>VoltDB</td>
<td>SERIALIZEABLE</td>
<td>SERIALIZEABLE</td>
</tr>
</tbody>
</table>

Source: Peter Bailis, *When is "ACID" ACID? Rarely*. January 2013
SQL-92 Access Modes

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

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

**Postgres + MySQL 5.6**
```
START TRANSACTION <access-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</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>Postgres</td>
<td>MVCC</td>
</tr>
<tr>
<td>MS SQL Server</td>
<td>Strict 2PL or MVCC</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>

Summary

- Concurrency control is hard.