A study of Concurrency Control Techniques in Main Memory Database Systems
Background

- Concurrency needed to mask I/O latency.
- It is experimentally proven that most time spent executing a transaction is used by components like buffer manager, lock manager and latching.
- In MMDB, no need to mask I/O misses.
- Hence, synchronization and concurrency components were removed for early memory database systems and all was well.
- **Then why concurrency now?**
Why concurrency for MMDB?

- Problem Area - **Partition Crossing Transactions**! even on a single node.
- Cannot use the good old concurrency control techniques, as they are heavy weight —> **light weight concurrency control mechanisms for MMDBs**.
Concurrency Control Types

• Pessimistic Concurrency
  • Assume conflicts will occur and avoids conflicts by acquiring locks on data that is being read or modified.
  • Readers and writers hence block each other.

• Optimistic Concurrency
  • Assume transactions are unlikely to modify data that another transaction is modifying.
  • Example - MVCC
Concurrency Control Techniques

- Partitioned Execution
- Timestamp Ordering
- Multi Version Concurrency Control (MVCC)
- Light Weight Locking
I. Partitioned Execution

- Multiple serial execution threads run in parallel for disjoint partitions of data.
- Either disallow the execution of transactions which might access more than one partition of the data.
  
  OR
  
- Acquire a database lock leading to serial execution mode without concurrency on separate partitions.

![Graph showing comparison of committed transactions per second (ctps) for partitioned and serial execution based on the number of partition crossing transactions in %]
II. Basic Timestamp Ordering

1. \( r_i(x) \): \( T_i \) wants to read \( x \):
   (a) If \( TS(T_i) < writeTS(x) \), the TO rule would be violated. Thus, the transaction \( T_i \) has to be aborted.
   (b) Otherwise, allow access and set \( readTS(x) := \max(TS(T_i), readTS(x)) \).
2. \( w_i(x) \): \( T_i \) wants to write \( x \):
   (a) If \( TS(T_i) < readTS(x) \) or \( TS(T_i) < writeTS(x) \),
       the TO rule would be violated. Thus, the transaction \( T_i \) has to be aborted.
   (b) Otherwise, allow access and set \( writeTS(A) := TS(T_i) \).

• Pros:
  • Generates serializable schedule.
  • Atomic Execution.

• Cons
  • Does not guarantee recoverability. \( \rightarrow \) dirty read.
II. Strict Timestamp Ordering
(Wolf et al. 2015)

• Concurrency
  • No uncommitted changes of a running transaction are overwritten or read by another transaction - done with use of dirty bit along with timestamp value.
  • Other transactions accessing the tuples with dirty bit need to wait.
  • Prevention of deadlocks by ensuring a transaction never waits for younger transaction.
  • Deleted tuples are marked with delete flag and deletion is deferred to the commit phase of a transaction. Other transactions accessing these tuples need to wait.

• Data structures
  • 32 bit Read write timestamp codes - includes dirty bit and delete flag.
  • Dirty Bit inventory.

• Synchronization
  • Use of MCS lock on index structures - uses spinning on thread-local variables for waiting and allows reader and writer synchronization.
  • When accessing a tuple from one partition - lock its index structures, looks up the tuple, perform admissibility check and access the tuple.
STO vs Serial Execution

Graph showing the comparison between STO and Serial Execution in terms of transactions per second (tps) with respect to the number of threads.

Bar chart illustrating the cycles count for read and write operations, with different percentages for each operation.
STO vs Partitioned

![Graph showing committed transactions per second (ctps) against the number of partition crossing transactions in % for different execution methods.]

- **Strict Timestamp Ordering**
- **Partitioned Execution**
- **Serial Execution**

TPC-C default
III. MVCC

- No update in place, each update creates a new version of that data object.
- Concurrent readers can still see the old version, while the update transaction proceeds concurrently.
- Read transactions never have to wait and with no use of locks.
- Assign Transaction ID and start timestamp to each transaction.
- Guarantees snapshot isolation.

- Drawbacks
  - Serializability guarantee is very expensive.
    - Known solutions require keeping track of the entire read set of every transaction.
  - Cascading Aborts.
III. Fast Serializable MVCC  
(Neumann et al. 2015)

- Idea
  - Update in-place.
    - Maintains contiguity of data vectors for high scan performance.
  - Use of undo-buffer to maintain previous versions.
    - No storage overhead.
  - Use of commit timestamp to determine their serializable order.
  - Read-only transactions read versions which are visible to them.
  - Update transaction go through a validation phase of only the predicate space.

- Visibility Check
  - \( v.p\text{pred} = \text{null} \lor v.p\text{pred}.TS = T \lor v.p\text{pred}.TS < T.\text{startTime} \)

- Avoid write-write conflicts
  - Prevents cascading aborts.
FS-MVCC vs SE vs MVCC

(a) HyPer with single-version concurrency control (■), our MVCC model (■), and a MVCC model that mimics the behavior of [23] by updating whole records and not using VersionedPositions (■).
Very LightWeight Locking
IV. Very Lightweight locking (Ren et al.)

- Avoid all overhead associated with traditional lock managers.
- Two changes:
  - Remove central locking and use tuple \((Cx, Cs)\) prepended with raw data
  - Request all locks at once and order the transactions in order of lock requests
- In high load, use Selective Contention Analysis.
- Implemented in Multi-partition, multi-worker thread model.
Very Lightweight locking

- Data structures: `TxnQueue` and `(Cx,Cs)`
- `Cx` = Exclusive lock `Cs` = Shared lock.
- Critical section (Execution of Tx)
  - Increment value by 1 for each request.
  - Transaction identifies all write and read sets in advance.
- Multi partition requests have to be handled using coordinators.

function FinishTransaction(Txn T)
    <begin critical section>
    foreach key in T.ReadSet
        data[key].Cs--;  
    foreach key in T.WriteSet
        data[key].Cx--;  
    TxnQueue.Remove(T);  
    <end critical section>
VLL Issues

- As TxnQueue size increases, contention increases.
- Algorithm puts an artificial limit on TxnQueue.
- Finish transaction before accepting new ones.
- Need of SCA.

<table>
<thead>
<tr>
<th>VLL</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Table" /></td>
<td><img src="image2.png" alt="Table" /></td>
</tr>
</tbody>
</table>

TransQueue
A, B, C, D
VLL Empirical analysis

Graph 1: Throughput (txns/sec) vs. contention index
- VLL with SCA
- VLL
- Standard 2PL
- Deadlock-free 2PL
- No locking

Graph 2: Throughput (txns/sec) vs. % multi-partition transactions
- VLL with SCA
- VLL
- Calvin
- H-Store
- Distributed 2PL
OCC in Multi-Core In-Memory DB
(Tu, Zheng et al.)

- Introduce Silo, designed for high performance in multicore systems
- Uses variant of OCC.
- OCC based commit protocol.

```c
    txn_commit() {
        // prepare commit
        commit_tid = atomic_fetch_and_add(&global_tid);
        // quickly serialize transactions
    }
```
Silo DB

- Provides txn serializability using Epochs.
- Assign eachtxn a *sequence number* and an *epoch*.
- Seq #s provide serializability during execution and epoch for recovery.
- Each record contains TID of its last writer.
- TID is broken into three pieces:

<table>
<thead>
<tr>
<th>Status bits</th>
<th>Sequence number</th>
<th>Epoch number</th>
</tr>
</thead>
</table>

OCC

- **Idea**: proceed as if records will not be modified – check otherwise at commit time.
- To read record A, save the record’s TID in a local *read-set*, then use the value.
- To write record A, store the write in a local *write-set*.
- In Silo:
  - **Phase 1**:
    - Lock (in global order) all records in the write set.
  - **Phase 2**:
    - Validate records in read set.
    - Abort if record’s TID changed or lock is held (by another transaction).
  - **Phase 3**:
    - Pick TID and perform writes.
    - Use the epoch recorded in Phase 1 for the TID.
Benefits of Silo

- Near linear scalability on popular database benchmarks.
Future Work and Reflection

• Improvement in latching overhead in STO.
• Evaluation of lock-free index structures.
• Compare concurrency control techniques used in commercial MMDBs such as SAP-Hana, Hekaton.
• Scenario Discussion for each type of concurrency control.
• Include more CC techniques.
Conclusion

• We covered Locking based, MVCC, timestamp ordering based concurrency control techniques.
• High Contention - VLL.
• Low Contention
  • OLTP
    • Perfectly partition-able workload - Partitioned Execution.
    • Partition-crossing transactions - STO.
    • Workload unknown - STO is a safe bet!
• OLTP and OLAP
  • Fast Serializable MVCC.