L-Store Concurrency Control: QueCC

Slides are adopted from Qadah, Sadoghi

*QueCC - A Queue-Oriented, Control-Free Concurrency Architecture*, ACM Middleware 2018

ECS 165A – Winter 2023
Hardware Trends

Large core counts

Large main-memory

HPE Superdome Server
144 physical cores
6TB of RAM

Popularity of Key-value Stores

- No multi-statement transactions
- Weak consistency
- Weak isolation
High-Contention Workloads

Challenge ???

High number of contented operations
State-of-the-Art Concurrency Control Protocols

- Optimized for multi-core hardware and main-memory databases
- Non-deterministic

<table>
<thead>
<tr>
<th>CC</th>
<th>Class</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>SILO</td>
<td>Optimistic CC</td>
<td>SOSP '13</td>
</tr>
<tr>
<td>TICTOC</td>
<td>Timestamp Ordering</td>
<td>SIGMOD '16</td>
</tr>
<tr>
<td>FOEDUS-MOCC</td>
<td>Optimistic CC</td>
<td>VLDB '16</td>
</tr>
<tr>
<td>ERMIA</td>
<td>MVCC</td>
<td>SIGMOD '16</td>
</tr>
<tr>
<td>Cicada</td>
<td>MVCC</td>
<td>SIGMOD '17</td>
</tr>
</tbody>
</table>
Optimize-for-multi-core concurrency control techniques suffer under high-contention due to increasing abort rate.
Performance Under High-Contention

Under high-contention: Non-deterministic aborts dominates
Performance Under High-Contention

Under high-contention: Non-deterministic aborts dominates
Client Transactions

2PL - NoWait

Abort Count: 0

each color presents a transaction
2PL - NoWait

Abort Count: 0

Client Transactions

- $w_4(b)$
- $w_3(b)$
- $r_4(d)$
- $r_3(c)$

Worker Thread #1

- $r_1(a)$
- $w_1(b)$

Worker Thread #2

- $w_2(b)$
- $r_2(a)$

Database:

- a
- b
- c
- d
2PL - NoWait

Abort Count: 0

Client Transactions

<table>
<thead>
<tr>
<th>w₄(b)</th>
<th>w₃(b)</th>
<th>r₄(d)</th>
<th>r₃(c)</th>
</tr>
</thead>
</table>

Worker Thread #1

| r₁(a) | w₁(b) |

Worker Thread #2

| w₂(b) | r₂(a) |

Database

| a | b | c | d |
Client Transactions

2PL - NoWait
Abort Count: 0

Worker Thread #1
- w1(b)
- r1(a)

Worker Thread #2
- w2(b)
- r2(a)

Conflict!
Client Transactions

\[ w_4(b) \quad w_3(b) \quad r_4(d) \quad r_3(c) \]

Worker Thread #1

\[ r_1(a) \quad w_1(b) \]

Worker Thread #2

\[ w_2(b) \quad r_2(a) \]

Abort transaction (to avoid potential deadlocks)

Abort Count: 0

2PL - NoWait
2PL - NoWait

Abort Count: 1

Client Transactions

Worker Thread #1

Worker Thread #2
2PL - NoWait

Abort Count: 1

Client Transactions

- $w_4(b)$
- $r_4(d)$
- $r_1(a)$
- $w_1(b)$

Worker Thread #1

- $w_3(b)$
- $r_3(c)$

Worker Thread #2

- $w_2(b)$
- $r_2(a)$

Database:

- a
- b
- c
- d
2PL - NoWait

Abort Count: 1

Client Transactions

- w₄(b)
- r₄(d)
- r₁(a)
- w₁(b)

Worker Thread #1
- w₃(b)
- r₃(c)

Worker Thread #2
- w₂(b)
- r₂(a)

Database:
- a
- b
- c
- d
2PL - NoWait

Abort Count: 1

Client Transactions
- w₄(b)
- r₄(d)
- r₁(a)
- w₁(b)

Worker Thread #1
- w₃(b)
- r₃(c)

Worker Thread #2
- w₂(b)
- r₂(a)

Conflict!
2PL - NoWait

Abort Count: 1

Abort transaction (to avoid potential deadlocks)

Client Transactions

- w₄(b)
- r₄(d)
- r₁(a)
- w₁(b)

Worker Thread #1

- w₃(b)
- r₃(c)

Worker Thread #2

- w₂(b)
- r₂(a)

Database:

- a
- b
- c
- d
2PL - NoWait

Abort Count: 2

Client Transactions

- \( w_4(b) \)
- \( w_3(b) \)
- \( r_1(a) \)
- \( r_4(d) \)
- \( r_3(c) \)
- \( w_1(b) \)

Worker Thread #1

Worker Thread #2

- \( w_2(b) \)
- \( r_2(a) \)
2PL - NoWait

Abort Count: 2

Client Transactions

- w_3(b)
- r_3(c)
- r_1(a)
- w_1(b)

Worker Thread #1

- w_4(b)
- r_4(d)

Worker Thread #2

- w_2(b)
- r_2(a)
2PL - NoWait

Abort Count: 2

Client Transactions
- $w_3(b)$
- $r_3(c)$
- $r_1(a)$
- $w_1(b)$

Worker Thread #1
- $r_4(d)$
- $w_4(b)$

Worker Thread #2
- $r_2(a)$
- $w_2(b)$
2PL - NoWait

Abort Count: 2

Client Transactions

\[ \begin{align*}
    w_3(b) & \quad r_3(c) \\
    r_1(a) & \quad w_1(b)
\end{align*} \]

Worker Thread #1

\[ \begin{align*}
    w_4(b) & \quad r_4(d)
\end{align*} \]

Worker Thread #2

\[ \begin{align*}
    w_2(b) & \quad r_2(a)
\end{align*} \]
2PL - NoWait

Abort Count: 2

Client Transactions

- $w_3(b)$
- $r_3(c)$
- $r_1(a)$
- $w_1(b)$

Worker Thread #1

- $w_4(b)$
- $r_4(d)$

Worker Thread #2

- $w_2(b)$
- $r_2(a)$

Conflict!
2PL - NoWait

Abort Count: 2

Abort transaction (to avoid potential deadlocks)

Client Transactions

w3(b)  r1(a)  w1(b)
r3(c)

Worker Thread #1
w4(b)  r4(d)

Worker Thread #2
w2(b)  r2(a)

a
b
c
d
2PL - NoWait

Abort Count: 3

Client Transactions

Worker Thread #1

Worker Thread #2

Committed Transactions

w₂(b), r₂(a)

a, b, c, d

w₄(b), w₃(b), r₁(a), w₁(b), r₄(d), r₃(c)
2PL - NoWait

Abort Count: 3

Client Transactions

Worker Thread #1

r₁(a)
w₁(b)
w₃(b)
r₃(c)

Worker Thread #2

w₄(b)
r₄(d)

Committed Transactions

w₂(b)
r₂(a)

Committed Transactions

a
b
c
d
2PL - NoWait

Abort Count: 3

Client Transactions

- \(r_1(a)\)
- \(w_1(b)\)

Worker Thread #1

- \(w_3(b)\)
- \(r_3(c)\)

Worker Thread #2

- \(w_4(b)\)
- \(r_4(d)\)

Committed Transactions

- \(w_2(b)\)
- \(r_2(a)\)

Database:

- \(a\)
- \(b\)
- \(c\)
- \(d\)
Worker Thread #1

\[ \begin{align*}
\text{r}_3(c) & \\
\text{w}_3(b) &
\end{align*} \]

Worker Thread #2

\[ \begin{align*}
\text{r}_4(d) & \\
\text{w}_4(b) &
\end{align*} \]

Client Transactions

\[ \begin{align*}
\text{r}_1(a) & \\
\text{w}_1(b) &
\end{align*} \]

Abort Count: 3

Committed Transactions

\[ \begin{align*}
\text{w}_2(b) \\
\text{r}_2(a) &
\end{align*} \]

2PL - NoWait

Conflict!
2PL - NoWait

Abort Count: 3

Client Transactions

Worker Thread #1

Worker Thread #2

Abort transaction (to avoid potential deadlocks)

Committed Transactions
2PL - NoWait

Abort Count: 4

Client Transactions

- $w_4(b)$
- $r_4(d)$
- $r_1(a)$
- $w_1(b)$

Committed Transactions

- $w_2(b)$
- $r_2(a)$

Worker Thread #1

- $w_3(b)$
- $r_3(c)$

Worker Thread #2

- $w_4(b)$
- $r_1(a)$
- $w_1(b)$

Database:

- $a$
- $b$
- $c$
- $d$
2PL - NoWait

Abort Count: 4

Client Transactions
- w₄(b)
- r₄(d)

Committed Transactions
- w₂(b)
- r₂(a)

Worker Thread #1
- w₃(b)
- r₃(c)

Worker Thread #2
- r₁(a)
- w₁(b)

Database:
- a
- b
- c
- d
2PL - NoWait

Abort Count: 4

Client Transactions

- $w_4(b)$
- $r_4(d)$

Worker Thread #1

- $w_3(b)$
- $r_3(c)$

Worker Thread #2

- $r_1(a)$
- $w_1(b)$

Committed Transactions

- $w_2(b)$
- $r_2(a)$
2PL - NoWait

Abort Count: 4

Client Transactions

- w₄(b)
- r₄(d)

Worker Thread #1

- w₃(b)
- r₃(c)

Worker Thread #2

- r₁(a)
- w₁(b)

Committed Transactions

- w₂(b)
- r₂(a)

Committed Transactions

- w₂(b)
- r₂(a)
2PL - NoWait

Abort Count: 4

Client Transactions

\[ \begin{align*}
    & w_4(b) \\
    & r_4(d)
\end{align*} \]

Worker Thread #1

\[ \begin{align*}
    & w_3(b) \\
    & r_3(c)
\end{align*} \]

Worker Thread #2

\[ \begin{align*}
    & r_1(a) \\
    & w_1(b)
\end{align*} \]

Committed Transactions

\[ \begin{align*}
    & w_2(b) \\
    & r_2(a)
\end{align*} \]
2PL - NoWait

Abort Count: 4

Client Transactions
- \( w_4(b) \)
- \( r_4(d) \)

Worker Thread #1
- \( w_3(b) \)
- \( r_3(c) \)

Worker Thread #2
- \( r_1(a) \)
- \( w_1(b) \)

Committed Transactions
- \( w_2(b) \)
- \( r_2(a) \)

Conflict!
2PL - NoWait

Abort Count: 4

Client Transactions
- \( w_4(b) \)
- \( r_4(d) \)

Worker Thread #1
- \( r_3(c) \)
- \( w_3(b) \)

Worker Thread #2
- \( r_1(a) \)
- \( w_1(b) \)

Abort transaction (to avoid potential deadlocks)

Committed Transactions
- \( w_2(b) \)
- \( r_2(a) \)
2PL - NoWait

Abort Count: 5

Client Transactions

- \( w_4(b) \)
- \( r_4(d) \)
- \( r_1(a) \)
- \( w_1(b) \)

Worker Thread #1

Worker Thread #2

Committed Transactions

- \( w_3(b) \)
- \( w_2(b) \)
- \( r_3(c) \)
- \( r_2(a) \)
2PL - NoWait

Abort Count: 5

Client Transactions

Worker Thread #1
- r1(a)
- w1(b)

Worker Thread #2
- w4(b)
- r4(d)

Committed Transactions
- w3(b), w2(b)
- r3(c), r2(a)
- Eventually transactions commit in some serial order!
- Many aborts due to high contention on record b
- Non-determinism in CC cause these aborts
- Wasted work
Key Insights

• Many aborts due to high contention

• Non-determinism in CC cause these aborts

• Can we do better?

• Is it possible to eliminate non-deterministic concurrency control from transaction execution?
Deterministic Transaction Execution

- H-Store [Kallman et al. ’08]
- Designed and optimized for horizontal scalability, multi-core hardware and in-memory databases
- Stored procedure transaction model
- Static partitioning of database
- Assigns a single core to each partition
- Execute transaction serially without concurrency control within each partition
Client Transactions

- $w_1(b)$
- $w_2(c)$
- $w_3(b)$
- $w_4(d)$
- $r_1(a)$
- $r_2(d)$
- $r_3(a)$
- $r_4(c)$

Single-partition transactions

Abort Count: 0

H-Store

Worker Thread #1

- P1
- P2

Worker Thread #2

P1 is assigned to Worker Thread #1

P2 is assigned to Worker Thread #2
Client Transactions

- $w_4(d)$
- $w_3(b)$
- $r_4(c)$
- $r_3(a)$

Abort Count: 0

H-Store

Worker Thread #1

- $r_1(a)$
- $w_1(b)$

Worker Thread #2

- $w_2(c)$
- $r_2(d)$

Committed Transactions

- P1
  - a
  - b
  - c
  - d

- P2
H-Store

Abort Count: 0

Client Transactions

Worker Thread #1
- $w_3(b)$
- $r_3(a)$

Worker Thread #2
- $w_4(d)$
- $r_4(c)$

Committed Transactions
- $w_2(c)$
- $r_1(a)$
- $r_2(d)$
- $w_1(b)$
H-Store

Abort Count: 0

Client Transactions

Worker Thread #1

Worker Thread #2

Committed Transactions

- w₄(d)
- w₃(b)
- w₂(c)
- r₁(a)
- r₄(c)
- r₃(a)
- r₂(d)
- w₁(b)
Client Transactions

Worker Thread #1

Worker Thread #2

H-Store

Abort Count: 0

Deterministic Execution
✓ No aborts because of CC
✓ Minimal coordination among threads

Performs well only when transactions are single-partitioned

Committed Transactions

w4(d) w3(b) w2(c) r1(a)
r4(c) r3(a) r2(d) w1(b)
Effect of Increasing Percentage of Multi-Partition Transactions in the Workload

H-Store is sensitive to the percentage of multi-partition transactions in the workload
Can We Do Better?

Our motivations are

• Efficiently exploits multi-core and large main-memory systems
• Provide serializable multi-statement transactions for key-value stores
• Scales well under high-contention workloads

Desired Properties

• Concurrent execution over shared data
• Not limited to partitionable workloads
• Without any concurrency controls
Is it possible to have concurrent execution over shared data without having any concurrency controls?
Introducing: QueCC
Queue-Oriented, Control-Free, Concurrency Architecture

A two parallel & independent phases of priority-driven planning & execution

**Phase 1:** Deterministic priority-based planning of transaction operations in parallel

- Plans take the form of *Prioritized Execution Queues*
- Execution Queues inherits predetermined priority of its planner
- Results in a deterministic plan of execution

**Phase 2:** Priority driven execution of plans in parallel

- Satisfies the *Execution Priority Invariance*

  “For each record (or a queue), operations that belong to higher priority queues (created by a higher priority planner) must always be executed before executing any lower priority operations.”
QueCC Architecture

Priority-based Parallel Planning Phase
QueCC Architecture

Priority-based Parallel Planning Phase

Batching Client Transactions

Planning Threads (Pre-determined Priority)

High Priority Queues

Low Priority Queues

Main Memory

DB Storage

Index
QueCC Architecture

Priority-based Parallel Planning Phase

Batching Client Transactions

Planning Threads
(Pre-determined Priority)

High Priority Queues
Low Priority Queues

Execution Queues

Main Memory
DB Storage

Index
QueCC Architecture

Queue-oriented Parallel Execution Phase

Batching Client Transactions

Planning Threads (Pre-determined Priority)

High Priority Queues

Low Priority Queues

Execution Queues

Execution Threads

Main Memory DB Storage

Index
Client Transactions

Planning
Thread #2

Client Transactions

Planning
Thread #1

Priority Groups

Low-priority
Queues

High-priority
Queues

Committed Transactions
Client Transactions

Planning Thread #1

Planning Thread #2

Committed Transactions

Abort Count: 0

QueCC

Priority Groups

Low-priority Queues

High-priority Queues

w_1(b)

r_1(a)

w_2(b)

r_2(a)

w_3(b)

r_3(c)

w_4(b)

r_4(d)
Client Transactions

Planning Thread #1
- r_4(d)
- r_2(a)

Planning Thread #2
- w_4(b)
- w_2(b)

Priority Groups

Low-priority Queues
- w_3(b)
- r_3(c)

High-priority Queues
- r_1(a)
- w_1(b)

Committed Transactions
- a
- b
- c
- d
Client Transactions

Planning Thread #1
- w2(b)
- r2(a)

Planning Thread #2
- w4(b)
- r4(d)

Priority Groups

High-priority Queues
- r1(a)
- w1(b)

Low-priority Queues
- w3(b)
- r3(c)

Committed Transactions
- a
- b
- c
- d

QueCC
Abort Count: 0
Client Transactions

Planning Thread #2

Planning Thread #1

Prioritized Execution Queues

Priority Groups

Low-priority Queues

High-priority Queues

Committed Transactions
Client Transactions

Priority Groups

Low-priority Queues
- \( w_4(b) \)
- \( w_3(b) \)
- \( r_3(c) \)
- \( r_4(d) \)

High-priority Queues
- \( r_2(a) \)
- \( r_1(a) \)
- \( w_2(b) \)
- \( w_1(b) \)

Committed Transactions
- a
- b
- c
- d

Abort Count: 0

Execution Thread #2

Execution Thread #1
Client Transactions

Execution Thread #1
- r1(a)
- r2(a)
- w1(b)
- w2(b)

Execution Priority Invariance

Priority Groups
- High-priority Queues
  - r2(a)
  - r1(a)
  - w2(b)
  - w1(b)

- Low-priority Queues
  - w4(b)
  - w3(b)
  - r3(c)
  - r4(d)

Committed Transactions
- a
- b
- c
- d

Abort Count: 0
Client Transactions

Execution Priority Invariance

Execution Thread #1

Execution Thread #2

Priority Groups

Low-priority Queues:
- w₄(b)
- w₃(b)
- r₃(c)
- r₄(d)

High-priority Queues:
- r₁(a)
- r₂(a)
- w₁(b)
- w₂(b)

Committed Transactions:
- a
- b
- c
- d

Abort Count: 0

QueCC
Client Transactions

Execution Priority Invariance

Execution Thread #1

Execution Thread #2

Abort Count: 0

Priority Groups

Low-priority Queues

High-priority Queues

Committed Transactions

Invariance

QueCC
Client Transactions

Execution Thread #1

Execution Thread #2

QueCC

Abort Count: 0

Priority Groups

Low-priority Queues

High-priority Queues

Committed Transactions

QueCC

Abort Count: 0

Execution
Thread #2

w4(b)

Client Transactions

Execution
Thread #1

w3(b)

r3(c)

r1(a)

w2(b)

r2(a)

w1(b)

a

b

c

d

w4(b)

r4(d)
Deterministic Execution
✓ No aborts because of CC
✓ Minimal coordination among threads
✓ Not sensitive to multi-partition transactions
✓ Exploits Intra-transaction parallelism

QueCC
Abort Count: 0

Priority Groups
Low-priority Queues
High-priority Queues

Committed Transactions
w_4(b) w_3(b) w_2(b) r_1(a)
r_4(d) r_3(c) r_2(a) w_1(b)
ResilientDB Blockchain Fabric

Application Layer / Testbed (YCSB, SYCSB, TPC-C Benchmarks)

Enable/Disable Secure Transactions

Concurrent Control Protocols (2PL, QueCC, 2VCC, DORA, MVCC, Timestamp, H-Store, NoWait, Silo, Foedus, MOCC, TicToc, Cicada)

Consensus Protocols (GeoBFT, PoE, RCC, Delayed Replication, ByShard, RingBFT, Zyzzyva, Bitcoin-NG, PoW, PBFT, RBFT)

Transaction Manager

Execution Threads

Commit Protocols: (Q-Store, 2PC, 3PC, Calvin, EasyCommit)

Message/IO Queues

Storage Layer: Lineage-based Storage Architecture

Indexes

Data

Fault-tolerant Distributed Transactions on Blockchain., S. Gupta, J. Hellings, M. Sadoghi

https://github.com/resilientdb/
https://resilientdb.com/
Evaluation Environment

Hardware
- Microsoft Azure instance with 32 core
- CPU: Intel Xeon E5-2698B v3
  - 32KB L1 data and instruction caches
  - 256KB L2 cache
  - 40MB L3 cache
- RAM: 448GB

Workload
- YCSB: 1 table, 10 operations, 50% RMW, Zipfian distribution
- TPCC: 9 tables, Payment and NewOrder, 1 Warehouse

Software
- Operating System: Ubuntu LTS 16.04.3
- Compiler: GCC with -O3 compiler optimizations
Effect of Varying Contention

- 5 write and 5 read operation per transaction
- 32 worker threads

Workload contention resiliency
Cache locality under high-contention

3.3x
Effect of Varying Worker Threads

- 5 write and 5 read operation per transaction
- Zipfian theta = 0.99

Avoiding thread coordination & eliminating all execution-induced aborts
Effect of Increasing Percentage of Multi-Partition Transactions in the Workload

![Graph showing the effect of increasing percentage of multi-partition transactions on throughput. The X-axis represents the percentage of multi-partition transactions, ranging from 0 to 100. The Y-axis represents throughput in million TPS, ranging from 0 to 4. The graph shows a downward trend as the percentage of multi-partition transactions increases.]
Effect of Increasing Percentage of Multi-Partition Transactions in the Workload

QueCC is not sensitive to multi-partitioning
TPC-C Results

1 Warehouse (highly contended workload)
50% Payment + 50% NewOrder transaction mix

QueCC can achieve up to 3x better performance on high-contention TPC-C workloads
QueCC Conclusions

✓ Efficient, parallel and deterministic in-memory transaction processing

✓ Eliminates almost all aborts by resolving transaction conflicts \textit{a priori}

✓ Works extremely well under high-contention workloads
What’s Next: Q-Store

QueCC

Multi-core Single-node

Q-Store

Partitioned on Distributed Cluster

Q-Store: Distributed, Multi-partition Transactions via Queue-oriented Execution and Communication., T. Qadah, S. Gupta, M. Sadoghi, EDBT 2020
What’s Next: Q-Store

Batching Client Transactions

Plan Local and Remote Execution Queues
What’s Next: Q-Store

Batching Client Transactions

Plan Local and Remote Execution Queues

Deliver Remote Execution Queues

Q-Store: Distributed, Multi-partition Transactions via Queue-oriented Execution and Communication., T. Qadah, S. Gupta, M. Sadoghi, EDBT 2020
What’s Next: Q-Store

Batching Client Transactions

Plan Local and Remote Execution Queues

Deliver Remote Execution Queues

Execute Queues

Q-Store: Distributed, Multi-partition Transactions via Queue-oriented Execution and Communication., T. Qadah, S. Gupta, M. Sadoghi, EDBT 2020
What’s Next: Q-Store

**QueCC**

- Multi-core
- Single-node

**Q-Store**

- Partitioned on Distributed Cluster

- Parallel and distributed
- Queue-oriented execution and communication
- Minimal coordination among nodes and threads
What’s Next: QBFT

QueCC
- Execution Queues
- Multi-core
- Single-node

Q-Store
- Partitioned on Distributed Cluster

QBFT
- Partitioned & Replicated

QBFT
What’s Next: QBFT

✓ Queue-oriented Byzantine Fault-Tolerance
✓ Resilient planning followed by resilient execution