High-Performance ACID via
Modular Concurrency Control

Chao Xie¹, Chunzhi Su¹, Cody Littley¹, Lorenzo Alvisi¹, Manos Kapritsos², Yang Wang³

(slides by Mrigesh)
TODAY’S READING

• Background and Motivation
• Callas’ Key Contributions
• Transaction Grouping
• Experiments and Evaluation
• Conclusions and “The Future”
THE PROBLEM

• The ACID paradigm offers an easy way to think about (and program applications involving) transactions

• However, performance is not a strong suit for ACID systems, especially when distributed (higher latency)

• This is the price of isolation: intermediate states of a transaction are hidden from other transactions
PREVIOUS SOLUTIONS

• Spanner, H-Store avoid 2PC for certain transactions

• A move towards BASE (Basic Availability, Soft-state & Eventual consistency)
  • e.g. Salt, by the same group, BASEified some transactions

• SDD-1 used statically-defined transaction classes, with fixed read/write sets

• Lynx and Sagas used SC-cycles to chop transactions
OBSERVATIONS

• Traditionally ACID guarantees are implemented uniformly to all transactions

• Conservatism guarantees correctness, but not performance

• Callas introduces “Modular Concurrency”, where a given isolation property is enforced at two-levels—within a group and (by extension) across groups
WHAT’S THE BIG IDEA?

• Modular Concurrency Control
  • Separation of concerns
  • Decouple ACID abstraction from the mechanism used to support it
  • General-purpose solution
KEY CONTRIBUTIONS

• Systematic analysis of “transaction-grouping”

• More aggressive use of traditional concurrency-boosting

• Runtime Pipelining- in-group mechanism to:
  • Allow concurrent execution of transactions based on real-time static analysis of an SC-graph
  • Guarantee atomicity while preventing Aborted Reads and avoiding enforcing rollback safety
AN ISOLATION REFRESHER

• **DSG**: A graph with nodes = committed transactions (T) and directed edges that indicate a scope for conflict between them
  • Read dependency: T_i installs a version x_i of object x. T_j reads x_i
  • Anti dependency: T_i reads a version x_k of object x. T_j installs next version of x.
  • Write dependency: T_i installs a version x_i of object x. T_j installs next version of x.
AN ISOLATION REFRESHER

• Circularity: The execution history contains a directed cycle

• Aborted Reads: A committed transaction $T_2$ reads some object modified by an aborted transaction $T_1$.

• Intermediate Reads: A committed transaction $T_2$ reads a version of an object $x$ written by another transaction $T_1$ that was not $T_1$’s final modification of $x$.

• Standard for serializability: Preventing these 3 states
ANALYSIS OF CALLAS

• Nexus Locks
• Automated Transaction Chopping
• Runtime Pipelining
• Implementation and Evaluation
NEXUS LOCKS

• Core of Callas’ concurrency control mechanism

• New type of lock
  • Regulates conflicts between transactions from different groups
  • Places no constraint on transactions within same group

• In some cases, the release of a lock can be delayed
CROSS-GROUP ISOLATION

Example 1: Naive handling of nexus locks does not prevent circularity

Example 2: Traditional locking prevents circularity

Example 3: Callas enforces $T_2$ to release nexus locks after $T_1$ to prevent circularity
INTRA-GROUP ISOLATION

• Secret Sauce that enables Callas’ performance gains

• Effective optimization within groups requires:
  • Appropriate grouping techniques that maximize the potential for concurrency
  • Identifying mechanisms to increase concurrency within a group
TRANSACTION CHOPPING

• Break transactions into constituent sub-transactions, which can be interleaved

• Analysis using an SC-graph:
  • Vertices = Candidate sub-transactions
  • S-edges = Undirected links within the same transaction
  • C-edges = Connected links between different transactions accessing the same object

• Need to ensure: rollback safety and prevention of SC-cycles
RUNTIME PIPELINING

- Operations within a transaction piece are only allowed to access read-write tables of the same rank.

- For any pair of pieces p1 and p2 of a given transaction that access read-write tables, if p1 is executed before p2, then p1 must access tables of smaller rank than p2.

(a) SC-cycle analysis cannot chop

(b) Runtime Pipelining
RUNTIME PIPELINING IN PRACTICE

STEP ONE
- Input: transactions with dependency information
- Build table dependency graph $G$
- Sort graph vertices and rank tables
- In each transaction, place operations that access tables of the same rank in the same piece

STEP TWO
- Build operation dependency graph
- For each transaction, sort vertices to order pieces
- Order operations within each piece according to their original order
CALLAS IN PRACTICE (STEP ONE)
CALLAS IN PRACTICE (STEP TWO)

- $T_1$: Skip (no read-only operation)
- $T_2$: 

```
R(D)  R(B)  W(A)  W(C)
```

- $T_1$: 

```
R(A)  R(B)  W(B)  W(C)
```

No crossing C-edges
EVALUATION : GOALS

• Compare performance improvement over equivalent ACID
• Compare performance improvement of each optimization
• Impact of different parameters / settings on performance
• Overhead of Nexus Locks
EVALUATION : TESTBED

• Three applications:
  • TPC-C
  • Fusion Ticket
  • Front Accounting

• Experimental setup:
  • MySQL Cluster
  • 10 database partitions
  • 3-way replicated
  • System saturated with load

• On Dell PowerEdge R320 machines
  • Xeon E5-2450 processor, 16 GB of memory, four 7200 RPM SATA disks, and 1 Gb Ethernet
• Callas performs:
  • 8.2x better on TPC-C
  • 6.7x better on Front Accounting
  • 5.7x better on Fusion Ticket
EVALUATION: INDIVIDUAL OPTIMIZATIONS

- Optimizations are App-dependent
- In TPC-C, simple choppings in the same group improve performance; FT benefits more from creating multiple groups
- In both cases, heuristically creating groups has a significant impact
EVALUATION : DIFFERENT GROUPINGS

- Simple runtime pipelining on a single group has a significant improvement in performance
- Naively breaking into 5 groups (one transaction per group) has a slight further increase
- Intelligent grouping to place frequently conflicting transactions in the same group has a 50% gain on TPC-C
In a micro-benchmark designed to neutralize Callas’ benefits, MySQL performs 19% better when no contention, and 13% better when contention is high.

Bottleneck is the increased CPU overhead of maintaining Nexus locks.

In high-contention, Callas’ message passing to enforce ordering in the bottleneck...
EVALUATION : CONTENTION RATE

- When inter-group contention is low, Callas >> MySQL Cluster
- Contention rate increases (due to more frequent transactions or more likely contention decreases this differential)
- Even in the worst case, Callas performs twice as well as the ACID MySQL Cluster
CONCLUSIONS AND FUTURE

• Callas relies on sound system design principles, and leverages smaller groups to improve performance.

• MySQL Cluster based prototype of Callas exhibits significant throughput gains.

• Need to verify performance gains on different systems (e.g. using OCC, MVCC) with different backends.