High Performance Transactions in Deuteronomy

Justin Levandoski, David Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang

Microsoft Research
Overview

**Deuteronomy:** componentized DB stack
- Separates transaction, record, and storage management
- Deployment flexibility, reusable in many systems and applications

Conventional wisdom: *layering incompatible with performance*

Build from the ground up for modern hardware
- Lock/latch-freedom, multiversion concurrency control,
  cache-coherence-friendly techniques

**Result:** *1.5M TPS*
- Performance rivaling in-memory database systems *but*
  clean separation & works even without in-memory data
The Deuteronomy Database Architecture

Transaction Control Component (TC)

- TC guarantees ACID
  - Logical concurrency control
  - Logical recovery
  - No knowledge of physical data storage

Data Component (DC)

- DC provides record storage
  - Physical data storage
  - Atomic record modifications
  - No knowledge of transactions, multiversioning

Record Operations (~CRUD)

Control Operations (Exactly Once, WAL, Checkpointing)
Deployment Flexibility

- Embeddable Key-Value Store
- Embeddable Transactional Store
- Networked Transactional Store
- Scale-out Transactional Store
- Fault-tolerant Scale-out Transactional Store
The First Implementation

Bottlenecked on locked remote ops

Operations per second

<table>
<thead>
<tr>
<th></th>
<th>TC</th>
<th>Bw-tree DC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>250×</td>
<td>100,000</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>1,000,000</td>
</tr>
<tr>
<td>100</td>
<td></td>
<td>10,000,000</td>
</tr>
<tr>
<td>1,000</td>
<td></td>
<td>100,000,000</td>
</tr>
<tr>
<td>10,000</td>
<td></td>
<td>1,000,000,000</td>
</tr>
<tr>
<td>100,000</td>
<td></td>
<td>10,000,000,000</td>
</tr>
<tr>
<td>1,000,000</td>
<td></td>
<td>100,000,000,000</td>
</tr>
</tbody>
</table>

Transactional Component (TC)

Record Manager

Log Manager

Lock Manager

Data Component (DC)
The New Transactional Component
Key Mechanisms for Millions of TPS

**Multiversion concurrency control (MVCC)**
Transactions never block one another
Multiversioning limited to TC only

**Lock and latch freedom throughout**
Buffer management, concurrency control, caches, allocators, ...

**In-memory recovery log buffers as version cache**
*Redo-only recovery* doubles in-memory cache density
Only committed versions sent to DC, shipped in log buffer units

**TC and DC run on separate sockets (or machines)**
Task parallelism/pipelining to gain performance
Data parallel when possible, but not at the expense of the user

- **Eliminate blocking**
- **Mitigate latency**
- **Maximize concurrency**
TC Overview

MVCC enforces serializability

Recovery log acts as version cache

Log buffers batch updates to DC

Parallel log replay engine at DC
Latch-free
Multiversion
Concurrency
Control
Timestamp MVCC

Each transaction has a timestamp assigned on begin
Transactions read, write, and commit at that timestamp

Each version marked with create timestamp
and last read timestamp
Latch-free MVCC Table

Records chained in hash table buckets

Hash Table

Key A | Version List | Read Time 40 | Key Y | Version List | Read Time 50

Version Manager
In-memory recovery log buffers + cache + DC

Create TxID 10 Log Offset
Create TxID 18 Log Offset
Create TxID 4 Log Offset
Latch-free MVCC Table

Ordered version lists chained off each record
Latch-free MVCC Table

**TxId gives version status and create timestamp**
Latch-free MVCC Table: Reads

Read: find a visible, committed version; compare-and-swap read timestamp
Latch-free MVCC Table: Reads

Data is pointed to directly in in-memory recovery log buffers

Hash Table

Key A | Version List | Read Time 40

Key Y | Version List | Read Time 50

Create TxID 10 | Log Offset

Create TxID 18 | Log Offset

Create TxID 4 | Log Offset

Version Manager

In-memory recovery log buffers + cache + DC
Latch-free MVCC Table: Reads

All metadata entries cacheline sized 6 cache misses in the common case
Work of indexing done by CC

Version Manager
In-memory recovery log buffers + cache + DC

Key A
Version List
Read Time 50

Key Y
Version List
Read Time 50

Hash Table

Miss
Miss
Miss
Miss
Miss
Miss

Create TxID 10 Log Offset
Create TxID 18 Log Offset
Create TxID 4 Log Offset

Version Manager

Miss
Latch-free MVCC Table: Writes

Append new version to in-memory log

Hash Table

Key A | Version List | Read Time 50

Key Y | Version List | Read Time 50

Create TxID 10 | Log Offset

Create TxID 18 | Log Offset

Create TxID 4 | Log Offset
Latch-free MVCC Table: Writes

Create new version metadata that points to it
Latch-free MVCC Table: Writes

Install version atomically with compare-and-swap
MVCC Garbage Collection

Track
   Oldest active transaction (OAT)
   Version application progress at the DC
Remove versions older than OAT and applied at the DC
Later requests for most recent version of the record go to DC
Latch-free Log Buffer Allocation
Serialized Log Allocation, Parallel Filling

Only allocation is serialized, not data copying
Fast Atomic Operations for Log Allocation

Thread 1: CompareAndSwap(&tail, 80, 90) → ok ✔️
Thread 2: CompareAndSwap(&tail, 80, 85) → fail ✗
Wasted shared-mode load for ‘pre-image’
Dilated conflict window creates retries

Thread 1: AtomicAdd(&tail, 10) → 90 ✔️
Thread 2: AtomicAdd(&tail, 5) → 95 ✔️
No need for load of ‘pre-image’
Order non-deterministic, but both succeed
TC Proxy

DC-side multicore parallel redo-replay
Multicore Replay at the DC

Each received log buffer replayed by dedicated hw thread

Fixed-size thread pool
Backpressure if entire socket busy

“Blind writes” versions to DC
“Delta chains” avoid read cost for writes

Out-of-order and redo-only safe
LSNs, only replay committed entries, shadow transaction table

Incoming Log Buffers from TC

TC Proxy

HW Threads

Data Component (Bw-tree)
Evaluation
Hardware for Experiments

4x Intel Xeon @ 2.8 GHz
64 total hardware threads

Commodity SSD ~450 MB/s
Experimental Workload

YCSB-like

50 million 100-byte values

4 ops/transaction

~“80-20” Zipfian access skew

DC on separate NUMA socket; also running periodic checkpoints

More than half of all records access every 20 seconds

Heavily stresses concurrency control and logging overheads
Evaluation: Transaction Throughput

84% reads
50% read-only transactions

1.5M TPS
Competitive w/ in-memory systems
Evaluation: Impact of Writes

- ~350,000 TPS w/100% writes
- Disk close to saturation
  - 90% disk bandwidth utilization
- DRAM latency limits write-heavy loads
  - More misses for DC update than for “at TC” read
For lack of time; fun stuff in the paper

Unapologetically racy log-structured read-cache

Fast async pattern
   Eliminates context switch and memory allocation overhead

Lightweight pointer stability
   Epoch protection for latch-free data structures free of atomic ops on the fast path

Fast commit with read-only transaction optimization

Recovery log as queue for durable commit notification

Thread management & NUMA details
Related Work

Modern in-memory database engines
  Hekaton [Diaconu et al]
  HANA
  HyPer [Kemper and Neumann]
  Silo [Tu et al]

Multiversion Timestamp Order [Bernstein, Hadzilacos, Goodman]

Strict Timestamp Order CC
  Hyper [Wolf et al]
Future Directions

Dealing with ranges
Timestamp concurrency control may be fragile
More performance work
More functionality
Evaluating scale-out
Conclusions

Deuteronomy: clean DB kernel separation needn’t be costly
Separated transaction, record, and storage management

Flexible deployment allows reuse in many scenarios
Embedded, classic stateless apps, large-scale fault-tolerant

Integrate the lessons of in-memory databases
Eliminate all blocking, locking, and latching
MVCC, cache-coherence-friendly techniques

1.5M TPS rivals in-memory database systems
but clean separation & works even without in-memory data