CS 764: Topics in Database Management Systems
Lecture 13: Modern OCC

Xiangyao Yu
10/19/2022
Announcement

Guest lecture next Monday (Oct. 24) in **virtual mode** (zoom only)
Project Idea Pitch

Problem Statement - Implement radix partitioned joins in a vectorized database engine.

Related work - Some key papers:
1. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory [Schuh et al.]
2. Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware [Balkesen et al.]
3. To Partition, or Not to Partition, That is the Join Question in a Real System [Bandle et al.]

Reach out to aaratik@cs.wisc.edu if interested!
Project Idea Pitch
Speedy Transactions in Multicore In-Memory Databases

Stephen Tu, Wenting Zheng, Edik Kohler1, Barbara Liskov, and Samuel Madden

MIT CSAIL and Harvard University

Abstract
Silo is a new in-memory database that achieves excellent performance and scalability on modern multicore machines. Silo was designing from the ground up to use system memory and caches efficiently. For instance, it avoids all conventional contention points, including that of centralized transaction ID assignment. Silo’s key contribution is a commit protocol based on optimistic concurrency control that provides serializability while avoiding all shared-memory writes for records that were only read. Though this might seem to complicate the enforcement of a serial order, correct logging and recovery is provided by linking periodically updated copies with the commit protocol. Silo provides the same guarantees as any serializable database without unnecessary scalability bottlenecks or much additional latency. Silo achieves almost 700,000 transactions per second on a standard TPC-C workload on a 12-core machine, as well as near-linear scalability. Considered per core, this is several times higher than previously reported results.

1 Introduction
Thanks to drastic increases in main memory sizes and processor core counts for server-class machines, modern high-end servers can have several terabytes of RAM and 80 or more cores. When used effectively, this is enough processing power and memory to handle data sets and computations that used to be spread across many disks and machines. However, harnessing this power is tricky; even single points of contention, like compare-and-swap on a shared-memory word, can limit scalability.

This paper presents Silo, a new in-memory database that achieves excellent performance on multicore machines. We designed Silo from the ground up to use system memory and caches efficiently. We avoid all centralized contention points and make all synchronizations scale with the data, allowing larger databases to support more concurrency.

Silo uses a Monroe-inspired 2PC structure for its underlying indexes. Monroe [23] is a fast concurrent BTtree-like structure optimized for multicore performance. But Monroe only supports non-serializable, single-key transactions, whereas any real database must support transactions that affect multiple keys and occur in some serial order. Our core result, the Silo commit protocol, is in a minimal-contention serializable commit protocol that provides these properties.

Silo uses a variant of optimistic, concurrency control (OCC) [18]. An OCC transaction reads the records it reads and writes in thread-local memory. At commit time, after validating that no concurrent transaction’s updates overlapped with its read set, the transaction installs all written records at once. If validation fails, the transaction aborts. This approach has several benefits for scalability: OCC writes to shared memory only on commit time, after the transaction’s compute phase has completed; this short write period reduces contention. And thanks to the validation step, read-set records need not be locked. This makes the memory writes required for read locality can instead contain contention [15].

Previous OCC implementations are not free of scaling bottlenecks, however, with a key issue being the requirement for tracking “anti-dependences,” i.e., artifact-read conflicts. Consider a transaction \( T \), that reads a record from the database, and a concurrent transaction \( T' \) that overwrites the value \( v \) assumed. A serializable system must order \( a \) before \( b \), even after a potential crash and recovery from persistent logs. To achieve this ordering, most systems require that \( T' \) communicate with \( T \), such as by posting its read set in shared memory or via a centrally assigned, monotonically-increasing transaction ID [19, 19]. Some non-serilizable systems can avoid this communication, but they suffer from anomalies like snapshot isolation’s “write skew” [2].

Applications can avoid scalability while avoiding all shared-memory writes for real transactions. The commit protocol was carefully designed using memory fences to scalability produce results consistent with a serial order. This incurs the problem of correct recovery, which we solve using a form of epoch-based garbage collection. Time is divided into a series of short epochs. Even though transactions always agree with a serial order, the system

SOSP, 2013
Outline

Multi-core scalability bottleneck

Silo OCC protocol
  – Read phase
  – Validation phase
  – Write phase

Discussion
  – Serializability proof sketch
  – Silo vs. OCC 1981
  – Phantom protection

OCC vs. 2PL
Even a single atomic instruction can become a scalability bottleneck.
Even a single atomic instruction can become a scalability bottleneck.

```
atomic_fetch_and_add(&lsn, size);
```

X. Yu et al. *Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores*, VLDB 2014
Silo Read Phase

Each tuple contains a 64-bit TID word

<table>
<thead>
<tr>
<th>Status bits</th>
<th>Sequence number</th>
<th>Epoch number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>63</td>
</tr>
</tbody>
</table>
Silo Read Phase

Each tuple contains a 64-bit TID word

<table>
<thead>
<tr>
<th>Status bits</th>
<th>Sequence number</th>
<th>Epoch number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>63</td>
</tr>
</tbody>
</table>

Each read returns consistent **value** and **TID word**

- Method 1: Guard the read with a latch (i.e., a short lock)
- Method 2: Optimistic lock (Silo’s approach)
Silo Read Phase

Each tuple contains a 64-bit TID word

<table>
<thead>
<tr>
<th>Status bits</th>
<th>Sequence number</th>
<th>Epoch number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>63</td>
<td></td>
</tr>
</tbody>
</table>

Each read returns consistent **value** and **TID word**
- Method 1: Guard the read with a latch (i.e., a short lock)
- Method 2: Optimistic lock (Silo’s approach)

```c
// read a record
do
    v1 = t.read_TID_word();
    RS[t.key].data = t.data
    v2 = t.read_TID_word();
while (v1 != v2 or v1.lock_bit == 1);
```
Silo Read Phase

Each tuple contains a 64-bit TID word

<table>
<thead>
<tr>
<th>Status bits</th>
<th>Sequence number</th>
<th>Epoch number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>63</td>
<td></td>
</tr>
</tbody>
</table>

Each read returns consistent **value and TID word**
- Method 1: Guard the read with a latch (i.e., a short lock)
- Method 2: Optimistic lock (Silo’s approach)

```c
// read a record
do
    v1 = t.read_TID_word();
    RS[t.key].data = t.data;
    v2 = t.read_TID_word();
while (v1 != v2 or v1.lock_bit == 1);

// write a record
v1.lock_bit = 1;
v1.update();
v1.update_seq_number();
v1.lock_bit = 0;
```
Silo Validation Phase

Data: read set $R$, write set $W$, node set $N$, global epoch number $E$

// Phase 1

for record, new-value in sorted($W$) do
    lock(record);
    compiler-fence();
    $e \leftarrow E$; // serialization point
    compiler-fence();

// Phase 2

for record, read-tid in $R$ do
    if record.tid $\neq$ read-tid or not record.latest
        or (record.locked and record $\notin W$)
        then abort();

for node, version in $N$ do
    if node.version $\neq$ version then abort();
    commit-tid $\leftarrow$ generate-tid($R$, $W$, $e$);

// Phase 3

for record, new-value in $W$ do
    write(record, new-value, commit-tid);
    unlock(record);
Silo Validation Phase

Phase 1: Lock the write set

Q: Why need to sort write set?

Data: read set \( R \), write set \( W \), node set \( N \),
global epoch number \( E \)

// Phase 1
for record, new-value in sorted(\( W \)) do
  lock(record);
  compiler-fence();
  \( e \leftarrow E; \) // serialization point
  compiler-fence();

// Phase 2
for record, read-tid in \( R \) do
  if record.tid \neq \text{read-tid} \text{ or not } record.\text{latest}
    \text{or } (record.\text{locked and record } \notin \text{\( W \)})
    \text{ then } \text{abort();}
for node, version in \( N \) do
  if node.version \neq \text{version} \text{ then } \text{abort();}
commit-tid \leftarrow \text{generate-tid}(R, W, e);

// Phase 3
for record, new-value in \( W \) do
  write(record, new-value, commit-tid);
  unlock(record);
Silo Validation Phase

**Phase 1: Lock the write set**

```
// Phase 1
for record, new-value in sorted(W) do
    lock(record);
compiler-fence();
e ← E;  // serialization point
compiler-fence();
```

**Phase 2: Validate the read set**

Validation fails if (1) the tuple is modified since the earlier read or (2) the tuple is locked by another transaction

```
// Phase 2
for record, read-tid in R do
    if record.tid ≠ read-tid or not record.latest
        or (record.locked and record ∉ W)
        then abort();
for node, version in N do
    if node.version ≠ version then abort();
commit-tid ← generate-tid(R, W, e);
// Phase 3
for record, new-value in W do
    write(record, new-value, commit-tid);
unlock(record);
```
Silo Validation Phase

Data: read set $R$, write set $W$, node set $N$, global epoch number $E$

// Phase 1
for record, new-value in sorted(W) do
  lock(record);
  compiler-fence();
  $e \leftarrow E$; // serialization point
  compiler-fence();
// Phase 2
for record, read-tid in $R$ do
  if record.tid $\neq$ read-tid or not record.latest or (record.locked and record $\not\in$ $W$)
    then abort();
for node, version in $N$ do
  if node.version $\neq$ version then abort();
  commit-tid $\leftarrow$ generate-tid($R$, $W$, $e$);
// Phase 3
for record, new-value in $W$ do
  write(record, new-value, commit-tid);
  unlock(record);

Phase 1: Lock the write set

Phase 2: Validate the read set
  - Validation fails if (1) the tuple is modified since the earlier read or (2) the tuple is locked by another transaction

Q: If a tuple is modified since a transaction’s earlier read, can the transaction still be serializable?
Silo Validation Phase

**Data:** read set $R$, write set $W$, node set $N$, global epoch number $E$

// Phase 1

```plaintext
for record, new-value in sorted($W$) do
    lock(record);
    compiler-fence();
    $e \leftarrow E$; // serialization point
    compiler-fence();
```

// Phase 2

```plaintext
for record, read-tid in $R$ do
    if record.tid $\neq$ read-tid or not record.latest
        or (record.locked and record $\notin W$)
        then abort();

for node, version in $N$ do
    if node.version $\neq$ version then abort();
    commit-tid $\leftarrow$ generate-tid($R$, $W$, $e$);
```

// Phase 3

```plaintext
for record, new-value in $W$ do
    write(record, new-value, commit-tid);
    unlock(record);
```

Phase 1: Lock the write set

Phase 2: Validate the read set

Phase 3: Write phase
Silo OCC is Serializable

read(A)  read(B)  read(C)  lock write set  validate read set  serialization point  write DB and release locks
Silo OCC is Serializable

Proof idea

– The Silo schedule is equivalent to an idealized schedule where all reads and writes of a transaction occur at the serialization point
– (Same strategy can be used to prove that 2PL is serializable)
// Phase 1
for record, new-value in sorted(W) do
    lock(record);
    compiler-fence();
    e ← E;                 // serialization point
    compiler-fence();
// Phase 2
for record, read-tid in R do
    if record.tid ≠ read-tid or not record.latest
        or (record.locked and record ∉ W)
        then abort();

tend = ( finish tn := tnc;
        valid := true;
        for t from start tn + 1 to finish tn do
            if (write set of transaction with transaction number t intersects read set)
                then valid := false;
            if valid
                then ((write phase); tnc := tnc + 1; tn := tnc));
        if valid
            then (cleanup)
            else (backup)).
Silo vs. OCC 1981

- Silo locks tuples in write set; OCC’81 uses global critical sections
Silo vs. OCC 1981

- Silo locks tuples in write set; OCC’81 uses global critical sections
- Silo validates using tuple versions; OCC’81 validates against write set of previous transactions

```plaintext
// Phase 1
for record, new-value in sorted(W) do
    lock(record);
    compiler-fence();
    e ← E; // serialization point
    compiler-fence();
// Phase 2
for record, read-tid in R do
    if record.tid ≠ read-tid or not record.latest
        or (record.locked and record ∉ W)
        then abort();

Silo
```
Silo vs. OCC 1981

• Silo locks tuples in write set; OCC’81 uses global critical sections
• Silo validates using tuple versions; OCC’81 validates against write set of previous transactions

Q: When is OCC 1981’s validation better than Silo’s validation?
Phantom Protection in 2PL

**Gap locks**

– A gap lock is a lock on a gap between index records, or a lock on the gap before the first or after the last index record (MySQL reference manual)
**Phantom Protection in 2PL**

**Gap locks**

- A gap lock is a lock on a gap between index records, or a lock on the gap before the first or after the last index record (MySQL reference manual)

```sql
SELECT *  
FROM table  
WHERE x > 6;
```
Phantom Protection in 2PL

**Gap locks**

- A gap lock is a lock on a gap between index records, or a lock on the gap before the first or after the last index record (MySQL reference manual)
- Next key lock = index node lock + gap lock before the record

```sql
SELECT * 
FROM table 
WHERE x > 6;
```
Phantom Protection in Silo

Data: read set $R$, write set $W$, node set $N$, global epoch number $E$

// Phase 1
for record, new-value in sorted($W$) do
  lock(record);
  compiler-fence();
  $e \leftarrow E$;  // serialization point
  compiler-fence();
// Phase 2
for record, read-tid in $R$ do
  if record.tid $\neq$ read-tid or not record.latest
    or (record.locked and record $\notin W$)
    then abort();
  for node, version in $N$ do
    if node.version $\neq$ version then abort();
    commit-tid $\leftarrow$ generate-tid($R$, $W$, $e$);
// Phase 3
for record, new-value in $W$ do
  write(record, new-value, commit-tid);
  unlock(record);

Validate the versions of accessed index nodes
- May need to consider the next nodes as well
Data: read set $R$, write set $W$, node set $N$, global epoch number $E$

// Phase 1
for record, new-value in sorted($W$) do
    lock(record);
    compiler-fence();
    e ← $E$;   // serialization point
    compiler-fence();

// Phase 2
for record, read-tid in $R$ do
    if record.tid ≠ read-tid or not record.latest
        or (record.locked and record $∉ W$)
        then abort();
for node, version in $N$ do
    if node.version $≠$ version then abort();
    commit-tid $←$ generate-tid($R$, $W$, $e$);

// Phase 3
for record, new-value in $W$ do
    write(record, new-value, commit-tid);
    unlock(record);

SELECT *
FROM table
WHERE x > 6;

3, 5
7
10, 13

Validate the versions of accessed index nodes
- May need to consider the next nodes as well
Discussions

Epochs in Silo: A mechanism to enable parallel logging
Discussions

**Epochs** in Silo: A mechanism to enable parallel logging

**Granularity of locking**: Support coarse-grained “locks” in Silo?
Discussions

**Epochs** in Silo: A mechanism to enable parallel logging

**Granularity of locking**: Support coarse-grained “locks” in Silo?

**Priority and preemption** of transactions?
Discussions

**Epochs** in Silo: A mechanism to enable parallel logging

**Granularity of locking**: Support coarse-grained “locks” in Silo?

**Priority** and **preemption** of transactions?

**Opacity**: Strict serializability for both committed and aborted transactions
  – Achieve opacity in 2PL vs. OCC?
Polaris

**Goal**: add priority mechanism to Silo

**Key idea**: add minimum pessimism into the protocol

- Transactions with higher priority can block transactions with lower priority
- Transactions within the same priority level run Silo
Polaris

**Goal**: add priority mechanism to Silo

**Key idea**: add minimum pessimism into the protocol
- Transactions with higher priority can block transactions with lower priority
- Transactions within the same priority level run Silo
Both Silo and Polaris achieve high throughput and low tail latency
Polaris

Read only workload

Both Silo and Polaris achieve high throughput and low tail latency

High-contention workload

Silo has decreased throughput and very high tail latency
- Some transactions experience repeated aborts

Polaris’ performance approaches 2PL at high contention
Q/A – Modern OCC

Silo applicable only to in-memory database?
How to achieve durability for in-memory database?
Extend Silo to a partitioned distributed system?
Modern systems using this concurrency control mechanism?
Support interactive query besides on-shot?
Global epoch number becomes a contention point?
Next Lecture

Guest lecture next Monday (Oct. 24) in **virtual mode** (zoom only)

Submit a review for the guest lecture
  – Deadline: **Oct. 28 (Friday), 11:59pm**
  – Use the same format as a paper review

Submit review before next Wednesday