Announcement

Guest lecture next Monday (Oct. 18) in **online mode**

**Round-table discussion** after the talk (2:00—3:00 PM)
Abstract

Silo is a new in-memory database that achieves excellent performance and scalability on modern multicore machines. Silo was designed from the ground up to use system memory and cache efficiently. For instance, it avoids all centralized contention points, including that of a 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. This might seem to complicate the enforcement of a serial order, conflict logging and recovery is provided by linking periodically-updated epochs 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 mix on a 32-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 cache efficiently. We avoid all centralized contention points and make all synchro-

nization scale with the data, allowing larger databases to support more concurrency.

Silo uses a Mazzer-inspired tree structure for its underlying index. Mazzer [23] is a fast concurrent B-tree-like structure optimized for multicore performance. But Mazzer 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 a minimal-commit serializable commit protocol that provides these properties.

Silo uses a variant of optimistic concurrency control (OCC) [18]. An OCC transaction tracks the records it reads and writes in thread-local storage. At commit time, after validating that no concurrent transaction’s writes 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 at commit time, after the transaction’s commit phase has completed; this short write period reduces contention. And thanks to the validation step, read-set records need not be locked. This matters because the memory writes required for read locks can induce contention [11].

Previous OCC implementations are not free of scaling bottlenecks, however, with a key reason being the requirement for tracking “anti-dependencies” (write-after-read conflicts). Consider a transaction $t_1$ that reads a record from the database, and a concurrent transaction $t_2$ that overwrites the value $v$; $t_1$ saw. A serializable system must order $t_1$ before $t_2$, even after a potential crash and recovery from persistent logs. To achieve this ordering, most systems require that $t_2$ communicate with $t_1$, such as by posting its read set to shared memory or via a centrally-located, monotonically-increasing transaction ID [18, 19]. Some non-serializable systems can avoid this communication, but they suffer from anomalies like snapshot isolation’s “write skew” [2].

Silo provides serializability while avoiding all shared-memory writes for read transactions. The commit protocol was carefully designed using memory fences to scalably produce results consistent with a serial order. This leaves the problem of correct recovery, which we solve using a form of epoch-based group commit. Time is divided into a series of short epochs. Even though transactions reach results always agree with a serial order, the system
Timestamp Allocation Bottleneck

Even a single atomic instruction can become a scalability bottleneck

```
atomic_fetch_and_add(&lsn, size);
```
Even a single atomic instruction can become a scalability bottleneck.
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></td>
<td>63</td>
</tr>
</tbody>
</table>
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></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
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 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 $\neq$ 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

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 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);
Silo Validation Phase

Phase 1: Lock the write set

Phase 2: Validate the read set

Phase 3: Write 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 ← 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);
Silo OCC is Serializable

lock write set
validate read set
write DB and release locks

read(A)  read(B)  read(C)  serialization point
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)
Silo vs. OCC 1981

// 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();

OCC 1981

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

```c
// 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 validates using tuple versions; OCC’81 validates against write set of previous transactions
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)

```
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

```
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
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);

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?
Evaluation

Figure 11: Effect of workload skew: 100 M records, 224 threads, payload 4 bytes, 10 ops/transaction.
Evaluation

Figure 6: YCSB with varying thread count, stored-procedure mode ($\theta = 0.9$, $read\_ratio = 0.5$)
Evaluation

Figure 6: YCSB with varying thread count, stored-procedure mode ($\theta = 0.9$, read\_ratio = 0.5)

Figure 7: YCSB with 5% long read-only transactions accessing 1000 tuples, stored-procedure mode ($\theta = 0.9$, read\_ratio = 0.5)
Q/A – Modern OCC

Is in-memory DB practical?
Source code of Silo and Masstree?
Silo with distributed databases?
Why need epochs?
Why need secondary index?
Example transactions that are not one shot?
Why use locks at all in an optimistic protocol?
Before Next Wednesday

Submit review for