Silo

New in-memory database that achieves excellent performance and scalability on modern multi-core machines.

Related work

Non-transactional systems

- Masstree, PALM
- Based on modified B-trees
- Fine-grained locking
- Techniques used in PLAM could further improve Silo
- Silo B-tree implementation inspired by Masstree

Fast but doesn't provide serialization.

Transactional systems

- Locking based systems
- Data partitioning
- Data duplication

Serializable transactions. Fast single-partition transactions.
Global contention points. Slow multi-partition transactions

Optimistic concurrency control:

1. Read and track required records
2. Compute
3. Try to commit changes
   - If no concurrent writes occurred: write changes
   - else abort

Read *doesn't* need locks. Short locking only during write.

Tracking required to find write-after-read conflicts.
Transactions can abort.

Silo - Introduction

- Relational database
- Efficient use of cache and system memory
- Avoids all *centralized* contention points
- Database stored in shared memory
- One-shot requests
- Commit protocol based on OCC
- Epoch based batch commit/logging
- Long lived recent consistent snapshots

Silo - Architecture and design

Silo tables are implemented as collections of index trees, one tree for each index.

Data is stored only in primary key tree, secondary key tries store primary key values.

Each one-shot request is dispatched to a single database worker thread.

Each one-shot request is dispatched to a single database worker thread.

Primary data is in memory, transactions are made durable by logging to disk.

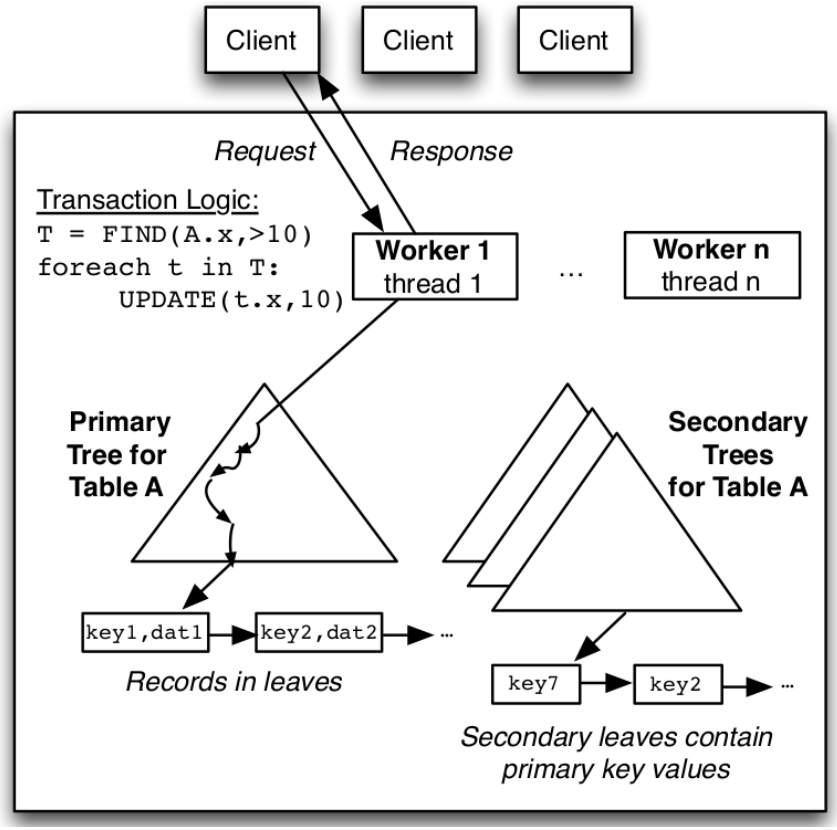

Figure: Silo architecture

Transaction IDs are 64 bit integers used to:

- Identify transactions
- Identify record versions
- Detect conflicts

Transaction ID format:

Epoch number	transaction number	status bits

Each record contains the TID of the transaction that most recently modified it. TIDs are assigned in decentralized fashion and are strictly increasing.

Records contain:

- TID of the transaction that most recently modified it
- Pointer to prevoius version of the record
- Record data

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*);

Read is is simple, no lock required. Only verify while committing if value is fresh

Write requires to lock overwritten records. Each write updates record value first then TID.

New records are created before transaction starts. Uninitialized records have *absent* status bit set and TID = 0.

Delete only set *absent* status bit. Record proper deletion is managed by garbage collector.

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*);

Records are removed when no worker would want to read it.

When record is set to be deleted it has assigned epoch when deletion should happen.

Removal can be performed by another task or simply but workers themselves.

Separate loggers threads perform logging.

Logging makes commit permanent.

Each value change is recorded separately.

Database is recovered from logs only to the point of last *finished* epoch.

Evaluation

Experiments where run on single machine with 4 8-core processors (Intel Xenon E7-4830) with HT disabled.

Each core has private 32KB L1 cache, private 256 KB L2 cache, 8 cores on single processor share 24MB of L3 cache.

Machine has 256GB of ram with 64GB per socket.

OS is Linux 3.2.0

Workers assigned to specific cores, one worker per core.

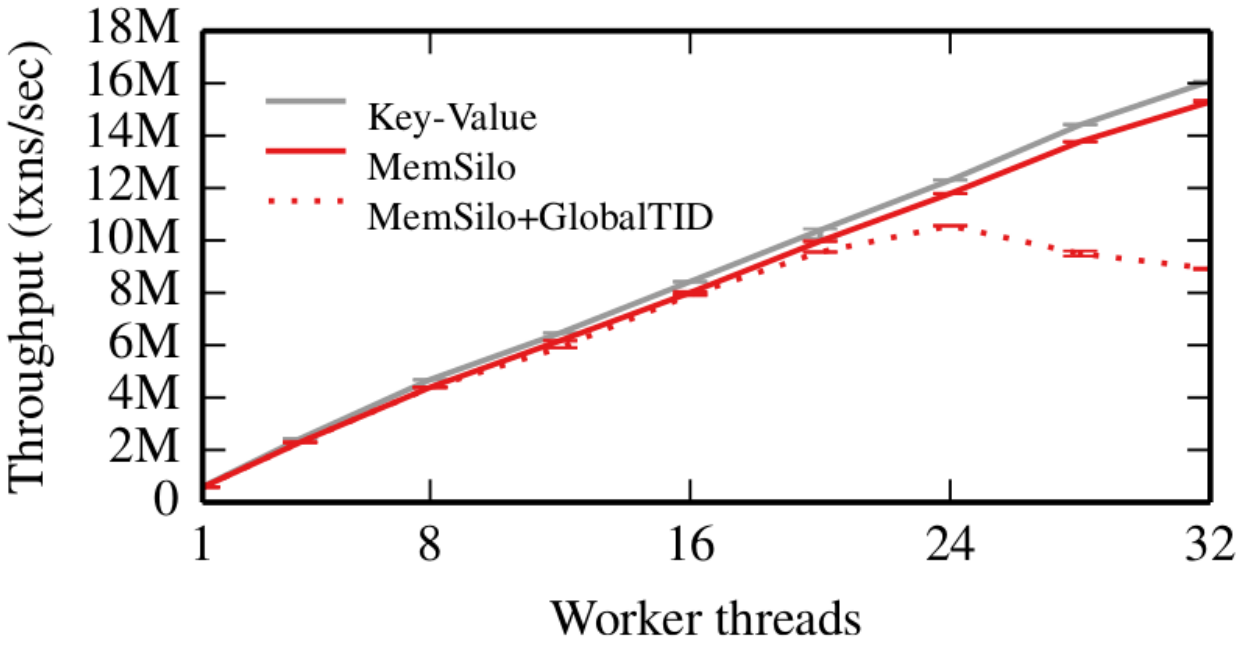

Figure 4: Overhead of *MemSilo* versus *Key-Value* when running a variant of the YCSB benchmark.

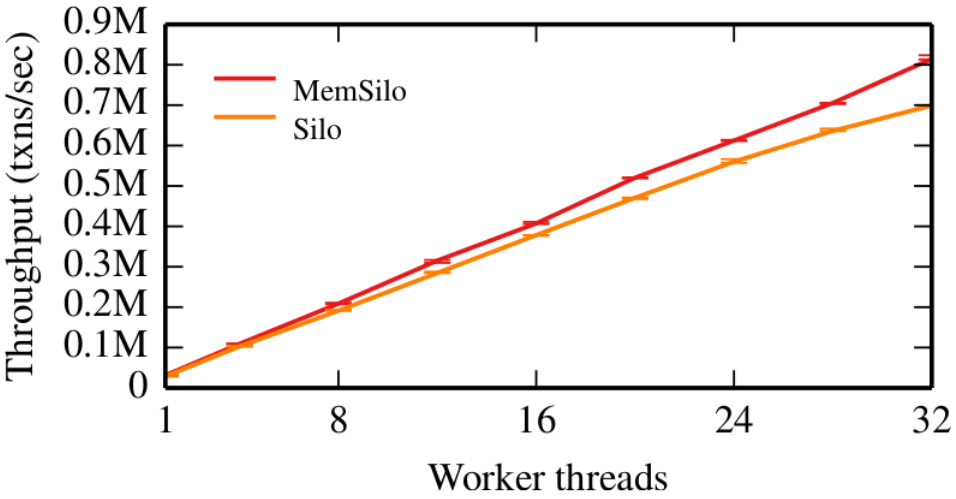

Figure 5: Throughput of Silo when running the TPC-C benchmark, including persistence.

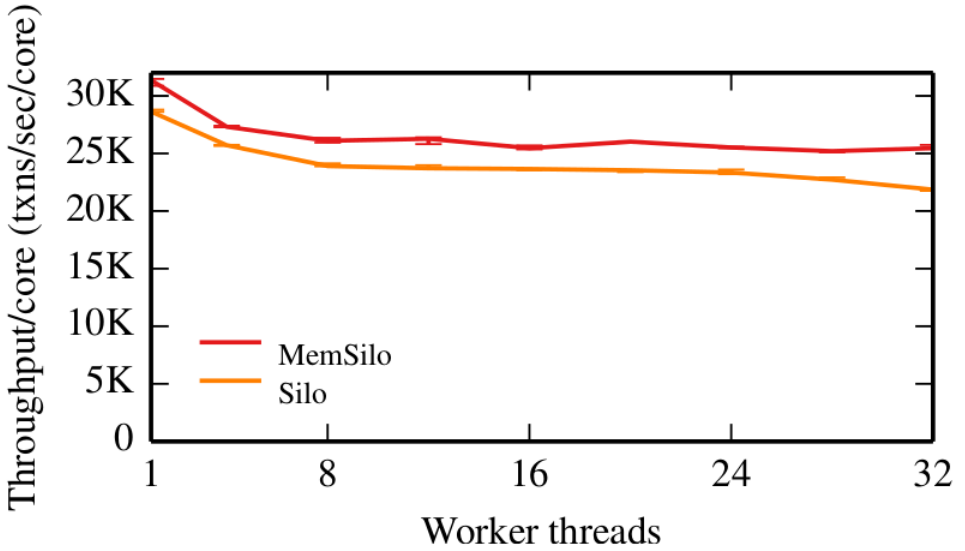

Figure 6: Per-core throughput of Silo when running the TPC-C benchmark.

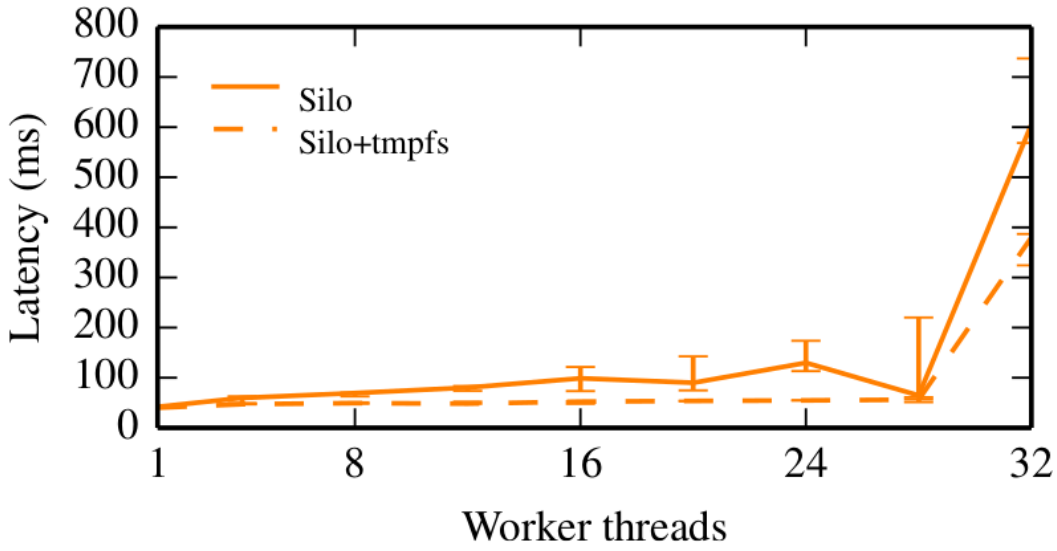

Figure 7: Silo transaction latency for TPC-C with logging to either durable storage or an in-memory file system.

	Transactions/sec	Aborts/sec
MemSilo	200,252	2,570
MemSilo+NoSS	168,062	15,756

Figure 10: Evaluation of Silo's snapshot transactions on a modified TPC-C benchmark.

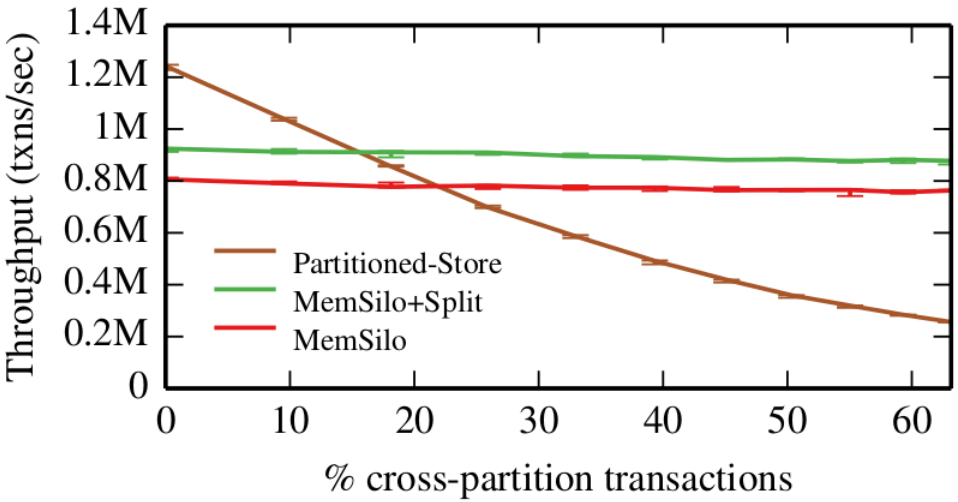

Figure 8: Performance of *Partitioned-Store* versus *MemSilo* as the percentage of cross-partition transactions is varied.

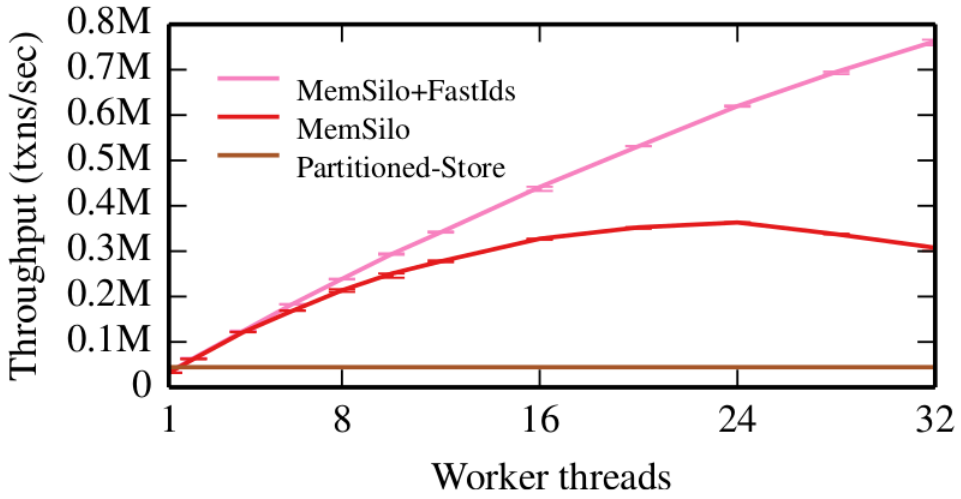

Figure 9: Performance of *Partitioned-Store* versus *MemSilo* as the number of workers processing the same size database is varied.

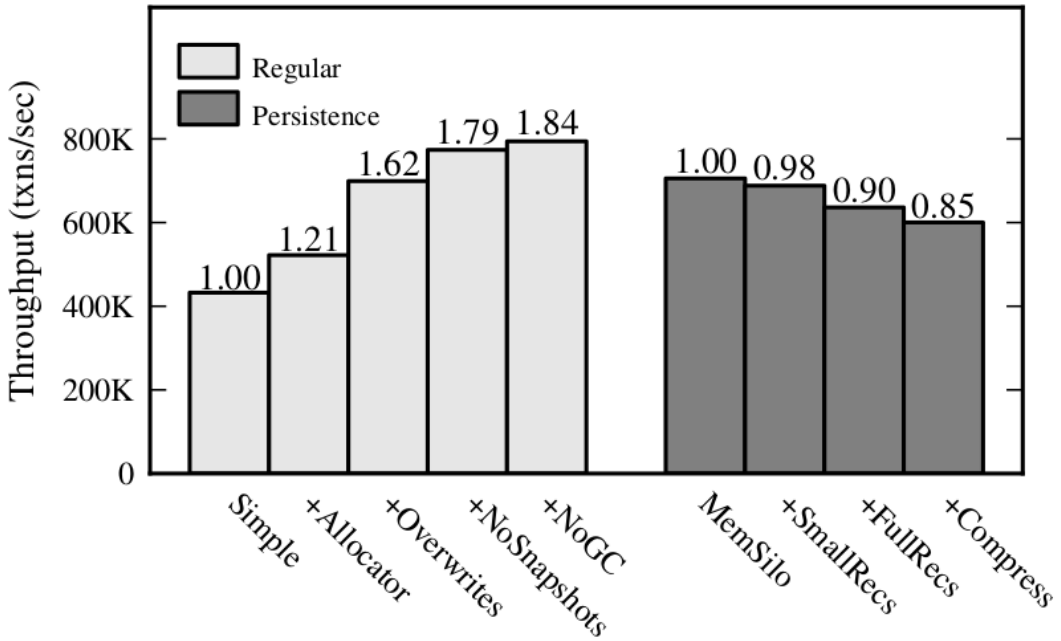

Figure 11: A factor analysis for Silo. Changes are added left to right for each group, and are cumulative.

Questions