Tabular: Efficiently Building Efficient Indexes

Ziyi Yan, Mohammed Farouk Drira, Tianxun Hu, Tianzheng Wang

SIMON FRASER UNIVERSITY

Concurrent data structure (index) is hard to build

- CPU/OS building blocks are hard to use
  - Synchronization primitives: latches, atomics
  - Locking protocol: pessimistic, optimistic, latch-free
  - Error-prone programming experience: deadlocks, memory leaks
- Need expertise in many neighboring areas
  - Memory management/Storage Devices/Networking
- Transaction semantics are easy to use
  - A transaction only need single-threaded logic
  - Concurrency control protocol guarantee ACID => Concurrent safety!
- DBMS engines have concurrency control but only for user transaction

Re-use *concurrency control* to implement *concurrent indexes*

Transactional Concurrent Programming

Transactional Concurrent Programming

- Start a new transaction

```
txn = BeginTransaction()
```

Transactional Concurrent Programming

- Start a new transaction

```
txn = BeginTransaction()
```

- Insert a new record

```
new_record =
        { name: "hand-crafted" }
rid = txn.Insert(table, record)
```

Transactional Concurrent Programming

- Start a new transaction

```
txn = BeginTransaction()
```

- Insert a new record

```
new_record =
       { name: "hand-crafted" }
rid = txn.Insert(table, record)
```

- Read the record by `rid`

```
record = txn.Read(table, rid)
```

Transactional Concurrent Programming

- Start a new transaction

```
txn = BeginTransaction()
```

- Insert a new record

```
new_record =
       { name: "hand-crafted" }
rid = txn.Insert(table, record)
```

- Read the record by `rid`

```
record = txn.Read(table, rid)
```

- Update the record by `rid`

```
record.name = "tabular"
txn.Update(table, rid, record)
```

Transactional Concurrent Programming

- Start a new transaction

```
txn = BeginTransaction()
```

- Insert a new record

```
new_record =
       { name: "hand-crafted" }
rid = txn.Insert(table, record)
```

- Read the record by `rid`

```
record = txn.Read(table, rid)
```

- Update the record by `rid`

```
record.name = "tabular"
txn.Update(table, rid, record)
```

- Commit the transaction

```
txn.Commit()
```

Transactional Concurrent Programming

- Start a new transaction

```
txn = BeginTransaction()
```

- Insert a new record

```
new_record =
       { name: "hand-crafted" }
rid = txn.Insert(table, record)
```

- Read the record by `rid`

```
record = txn.Read(table, rid)
```

- Update the record by `rid`

```
record.name = "tabular"
txn.Update(table, rid, record)
```

- Commit the transaction

```
txn.Commit()
```

Example: Atomically increment an integer

```
int COUNT;

void foo() {
  COUNT++;
}
```

Transactional Concurrent Programming

- Start a new transaction

```
txn = BeginTransaction()
```

- Insert a new record

```
new_record =
        { name: "hand-crafted" }
rid = txn.Insert(table, record)
```

- Read the record by `rid`

```
record = txn.Read(table, rid)
```

- Update the record by `rid`

```
record.name = "tabular"
txn.Update(table, rid, record)
```

- Commit the transaction

```
txn.Commit()
```

Example: Atomically increment an integer

```
int COUNT;

void foo() {
  COUNT++;
}
```

➡

```
std::mutex mutex;

void foo_lock() {
  mutex.lock();
  COUNT++;
  mutex.unlock();
}
```

Transactional Concurrent Programming

- Start a new transaction

```
txn = BeginTransaction()
```

- Insert a new record

```
new_record =
        { name: "hand-crafted" }
rid = txn.Insert(table, record)
```

- Read the record by `rid`

```
record = txn.Read(table, rid)
```

- Update the record by `rid`

```
record.name = "tabular"
txn.Update(table, rid, record)
```

- Commit the transaction

```
txn.Commit()
```

Example: Atomically increment an integer

```
int COUNT;

void foo() {
  COUNT++;
}
```

```
std::mutex mutex;

void foo_lock() {
  mutex.lock();
  COUNT++;
  mutex.unlock();
}
```

```
void foo_tabular() {

  cnt = txn.Read(tbl, cnt_id);
  cnt++;
  txn.Update(tbl, cnt_id, cnt);

}
```

Transactional Concurrent Programming

- Start a new transaction

```
txn = BeginTransaction()
```

- Insert a new record

```
new_record =
        { name: "hand-crafted" }
rid = txn.Insert(table, record)
```

- Read the record by `rid`

```
record = txn.Read(table, rid)
```

- Update the record by `rid`

```
record.name = "tabular"
txn.Update(table, rid, record)
```

- Commit the transaction

```
txn.Commit()
```

Example: Atomically increment an integer

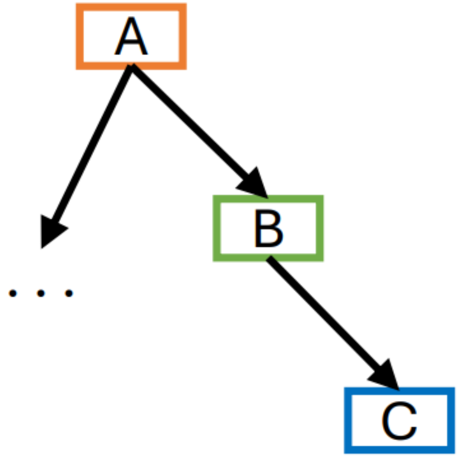
```
int COUNT;

void foo() {
  COUNT++;
}
```

```
std::mutex mutex;

void foo_lock() {
  mutex.lock();
  COUNT++;
  mutex.unlock();
}
```

```
void foo_tabular() {
  txn = BeginTransaction()
  cnt = txn.Read(tbl, cnt_id);
  cnt++;
  txn.Update(tbl, cnt_id, cnt);
  txn.Commit()
}
```

B+-tree on transactional tables

- Each node is stored as a record on table
- Each field in struct definition of `BTreeNode` becomes a column in schema definition
- Nodes are **"connected" via RIDs** instead of pointers



(a) Logical view

```
struct BTreeNode {
  int capacity;
  int n_keys;
  bool is_leaf;
  int lock;
  BTreeNode *right_child;
  KVPair kvs[16];
};
```

(b) C/C++ struct/class definition

RID	n_keys	is_leaf	right_child	kvs
0	10	False	1	...
1	8	False	2	...
2	4	True	INVALID_RID	...
...				

(c) Table storing BTreeNode objects

Tabular vs Hand-crafted B+-tree lookup()

Single-threaded code

```
def BTree::lookup(Key k, Value *out_v):
  n = root
  while n is inner:
    next = n.findChild(n, k)
    n = next
  *out_v = n.findValue(k)
```

Tabular vs Hand-crafted B+-tree lookup()

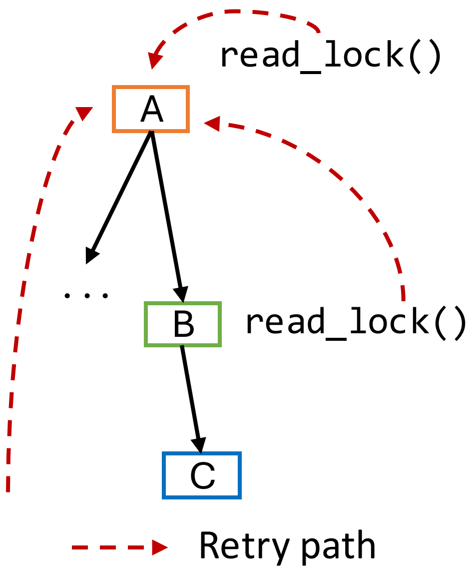
Single-threaded code

```
def BTree::lookup(Key k, Value *out_v):
  n = root
  while n is inner:
    next = n.findChild(n, k)
    n = next
  *out_v = n.findValue(k)
```

Hand-crafted code using Optimistic Lock Coupling (OLC)

Epoch manager implementation ...

Optimistic lock implementation...




```
def BTree::lookup(Key k, Value *out_v):
restart:
epoch_enter()
retry = false
  n = root
  ver = n.read_lock(retry)
  if retry or n != root: goto restart
  while n is inner:
    next = n.children[findChild(n, k)]
    n.verify_read(ver, retry)
    if retry: goto restart
    ver_next = next.read_lock(retry)
    if retry: goto restart
    n = next
  *out_v = n.findValue(k)
epoch_exit()
```

Tabular vs Hand-crafted B+-tree lookup()

Single-threaded code

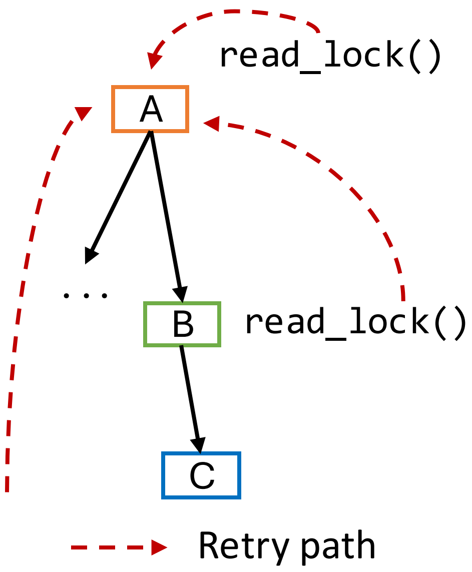
```
def BTree::lookup(Key k, Value *out_v):
  n = root
  while n is inner:
    next = n.findChild(n, k)
    n = next
  *out_v = n.findValue(k)
```

Tabular-based code

```
def BTree::lookup(Key k, Value *out_v):
  t = BeginTransaction()
  n = t.Read(table, root_id)
  while n is inner:
    next_id = n.findChild(n, k)
    n = t.Read(table, next_id)
  *out_v = n.findValue(k)
  t.Commit()
```

Hand-crafted code using Optimistic Lock Coupling (OLC)

Epoch manager implementation ...




```
def BTree::lookup(Key k, Value *out_v):
restart:
epoch_enter()
retry = false
  n = root
  ver = n.read_lock(retry)
  if retry or n != root: goto restart
  while n is inner:
    next = n.children[findChild(n, k)]
    n.verify_read(ver, retry)
    if retry: goto restart
    ver_next = next.read_lock(retry)
    if retry: goto restart
    n = next
  *out_v = n.findValue(k)
epoch_exit()
```

Optimistic lock implementation...

Tabular vs Hand-crafted B+-tree lookup()

Single-threaded code

```
def BTree::lookup(Key k, Value *out_v):
  n = root
  while n is inner:
    next = n.findChild(n, k)
    n = next
  *out_v = n.findValue(k)
```

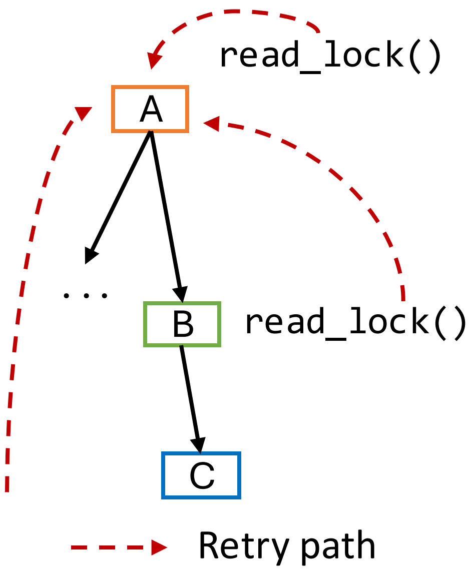
Tabular-based code

```
def BTree::lookup(Key k, Value *out_v):
  t = BeginTransaction()
  n = t.Read(table, root_id)
  while n is inner:
    next_id = n.findChild(n, k)
    n = t.Read(table, next_id)
  *out_v = n.findValue(k)
  t.Commit()
```

Hand-crafted code using Optimistic Lock Coupling (OLC)

Epoch manager implementation ...

Optimistic lock implementation...




```
def BTree::lookup(Key k, Value *out_v):
restart:
epoch_enter()
retry = false
  n = root
  ver = n.read_lock(retry)
  if retry or n != root: goto restart
  while n is inner:
    next = n.children[findChild(n, k)]
    n.verify_read(ver, retry)
    if retry: goto restart
    ver_next = next.read_lock(retry)
    if retry: goto restart
    n = next
  *out_v = n.findValue(k)
epoch_exit()
```

Easy to develop, debug and maintain

Lots of branches, massive mental overhead

Tabular vs Hand-crafted B+-tree lookup()

Single-threaded code

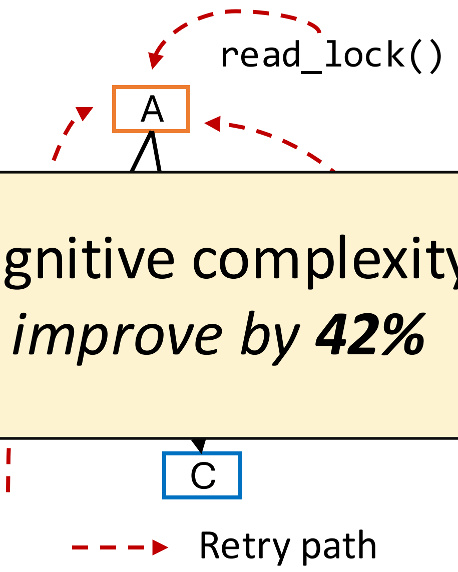
```
def BTree::lookup(Key k, Value *out_v):
  n = root
  while n is inner:
    next = n.findChild(n, k)
    n = next
  *out_v = n.findValue(k)
```

Tabular-based code

```
def BTree::lookup(Key k, Va
  t = BeginTransaction()
  n = t.Read(table, root_id)
  while n is inner:
    next_id = n.findChild(n, k)
    n = t.Read(table, next_id)
  *out_v = n.findValue(k)
  t.Commit()
```

Hand-crafted code using Optimistic Lock Coupling (OLC)

Epoch manager implementation ...




```
def BTree::lookup(Key k, Value *out_v):
restart:
epoch_enter()
retry = false
  n = root
  ver = n.read_lock(retry)
  if retry or n != root: goto restart
    ile n is inner:
    next = n.children[findChild(n, k)]
    n.verify_read(ver, retry)
    if retry: goto restart
    ver_next = next.read_lock(retry)
    if retry: goto restart
    n = next
  *out_v = n.findValue(k)
epoch_exit()
```

Optimistic lock implementation...

Measuring cognitive complexity [1]:

*Tabular improve by **42%***

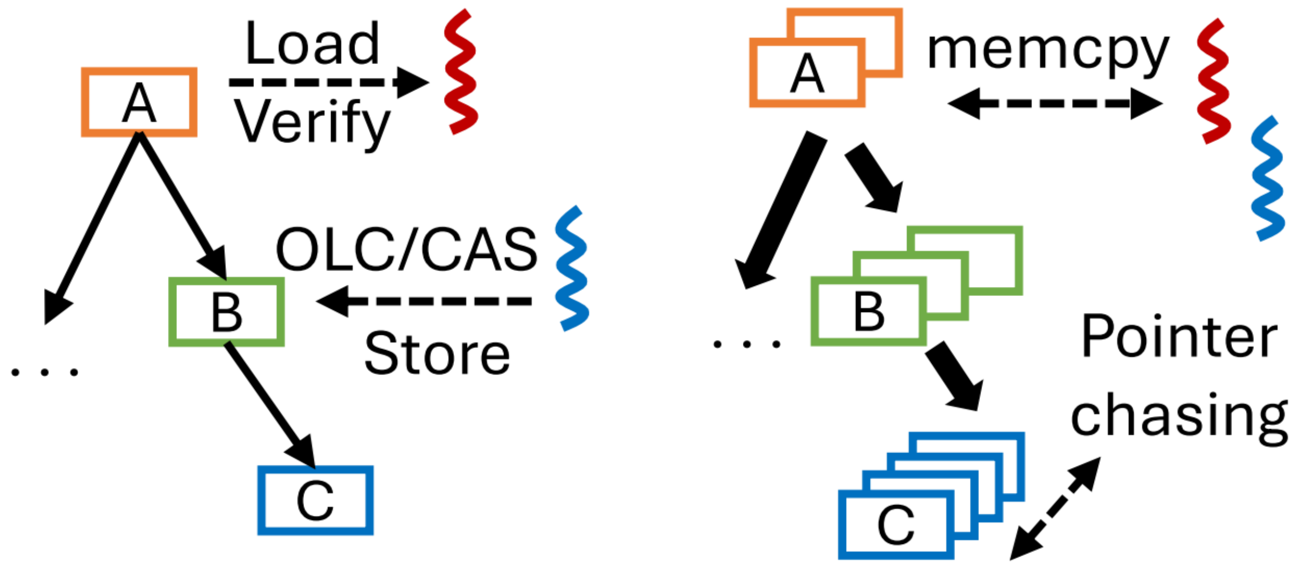
Easy to develop, debug and maintain

Lots of branches, massive mental overhead

[1]: G Ann Campbell. 2018. Cognitive complexity: An overview and evaluation. In Proceedings of the 2018 international conference on technical debt. 57–58.

Naïve objects-on-DB performance is bad

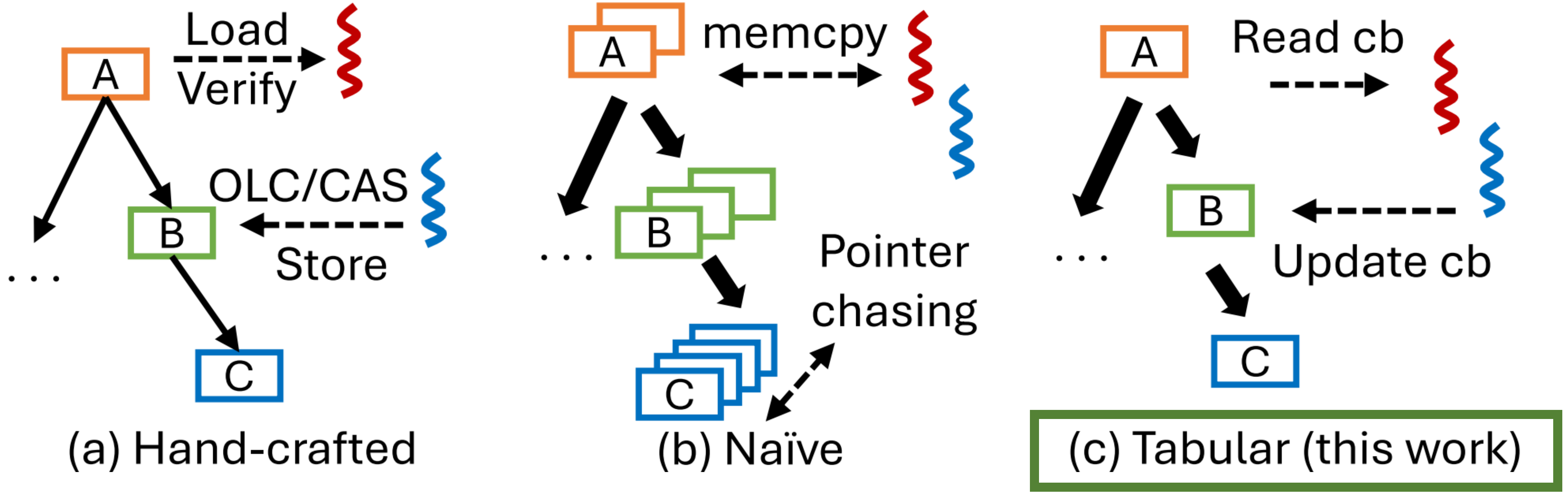
- Traditional concurrency control (CC) scheme incurs unnecessary overhead
  - Optimistic CC (OCC) needs memory copying to transaction-local memory
  - Multi-versioned CC (MVCC) stalls at pointer-chasing while accessing records



(a) Hand-crafted (b) Naïve

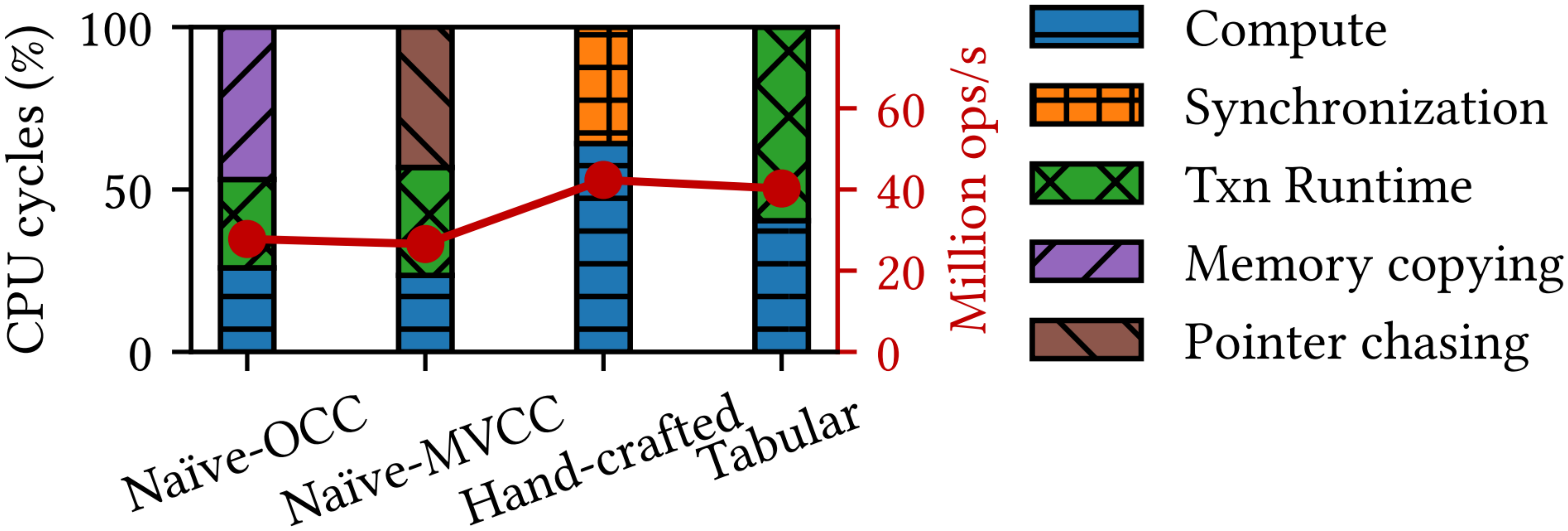
Naïve objects-on-DB performance is bad

- Traditional concurrency control (CC) scheme incurs unnecessary overhead
  - Optimistic CC (OCC) needs memory copying to transaction-local memory
  - Multi-versioned CC (MVCC) stalls at pointer-chasing while accessing records
- Tabular (this work) adopts: (1) single-versioned OCC for direct record access and (2) zero-copy reads and in-place updates via callback functions



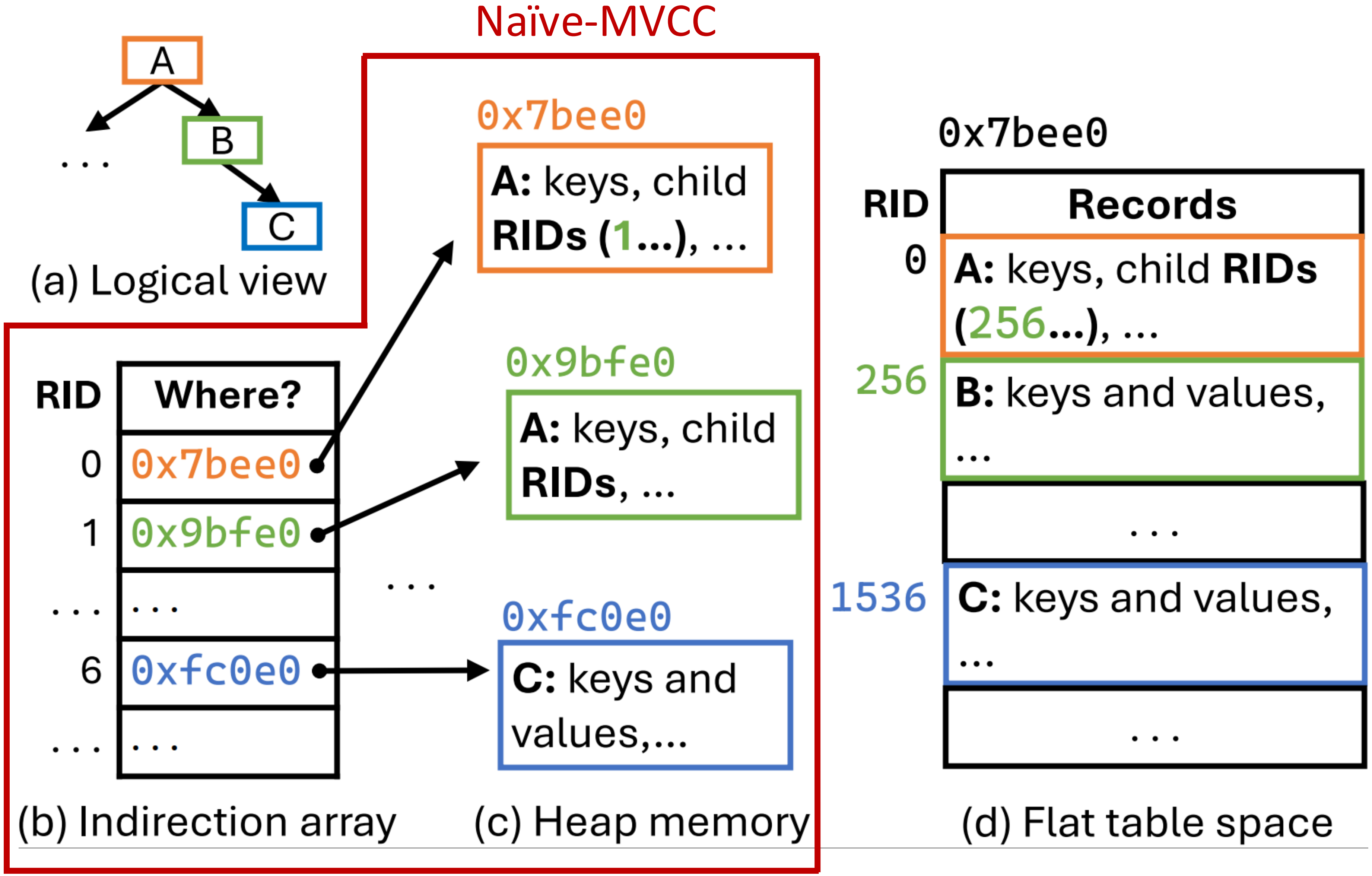
Naïve objects-on-DB cycle breakdowns

- Unnecessary overhead: Memory copying, Pointer chasing
- They took nearly 50% of cycles



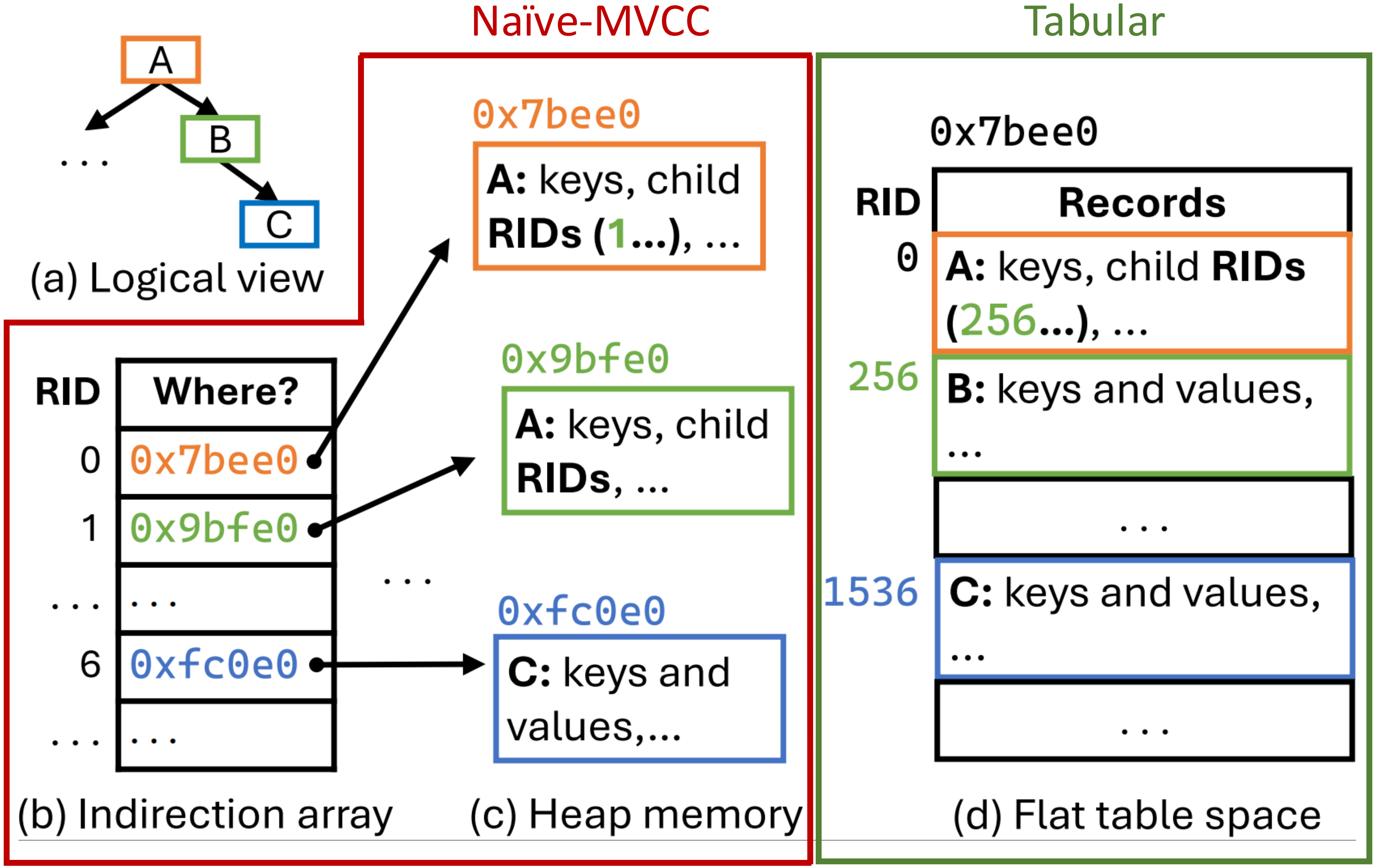
Tabular: Single-Versioned, Direct Record Access

- RID is the offset from base pointer of flat memory
- Records are stored directly on the table instead of referenced via pointers
- It **saves ~19-29%** cycles in B+-tree



Tabular: Single-Versioned, Direct Record Access

- RID is the offset from base pointer of flat memory
- Records are stored directly on the table instead of referenced via pointers
- It **saves ~19-29%** cycles in B+-tree



Tabular: Copy-based vs Callback-based interface

- OCC requires memory copying from tables to transactions

```
# copy-based
node = txn.Read(table, rid)
is_full = node.num_entries == MAX_ENTRIES

# callback-based
var is_full: bool
def callback(record):
  is_full = node.num_entries == MAX_ENTRIES
txn.ReadCallback(table, rid, callback)
```

SFU SIMON FRASER UNIVERSITY

Tabular: Zero-Copy Record Access

- OCC requires memory copying from tables to transactions
- Observation: Record access can be re-tried cheaper compared to memory copy
- User-defined callback executes directly on the records
- It saves ~20-31% cycles in B+-tree

```
def Transaction::ReadCallback(table, rid, callback):
  # Take a reference (pointer) to the record
  Record &record = table.GetRecord(rid);
retry:
  tid = record.get_consistent_tid()
  callback(record) # Execute the callback in-place

  # Verify the record did not change
  tid_now = record.get_consistent_tid()
  if (tid_now == tid):
    read_set.Add(record, tid);
  else:
    goto retry
```

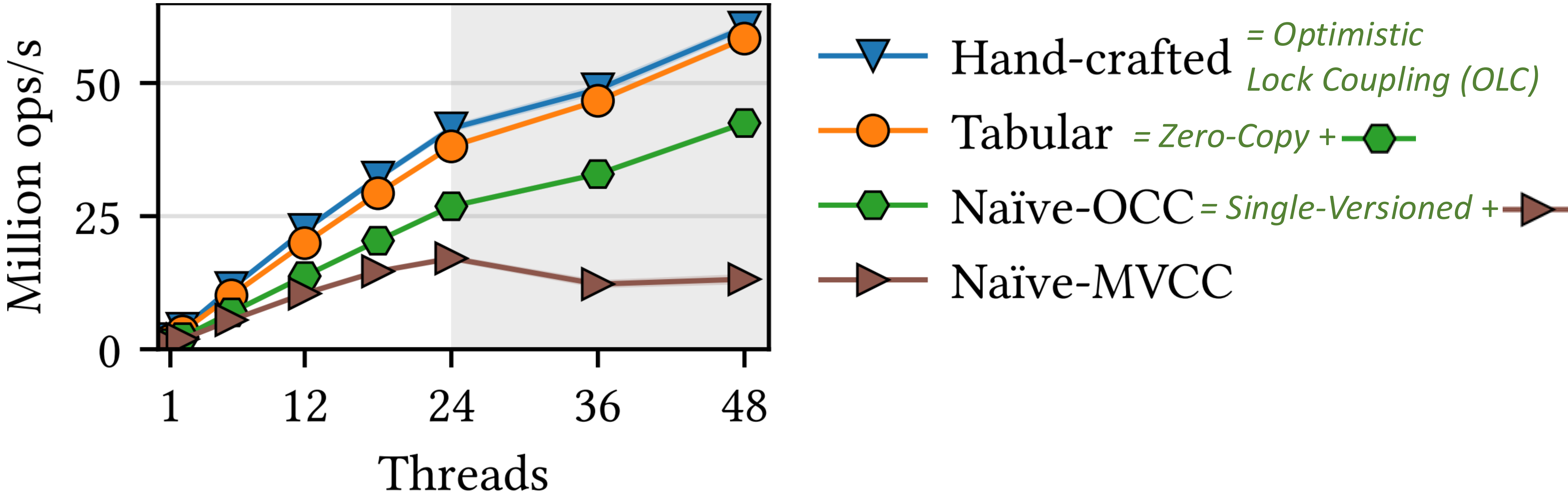
Tabular: Zero-Copy Record Access

- OCC requires memory copying from tables to transactions
- Observation: Record access can be re-tried cheaper compared to memory copy
- User-defined callback executes directly on the records
- It saves ~20-31% cycles in B+-tree

```
def Transaction::ReadCallback(table, rid, callback):
  # Take a reference (pointer) to the record
  Record &record = table.GetRecord(rid);
retry:
  tid = record.get_consistent_tid()
  callback(record) # Execute the callback in-place

  # Verify the record did not change
  tid_now = record.get_consistent_tid()
  if (tid_now == tid):
    read_set.Add(record, tid);
  else:
    goto retry
```

Tabular achieves competitive performance

- Workload: 100M of 8B-key B+-tree running 50% lookups and 50% updates
- Tabular performs at ~90% of Hand-crafted (Upper bound)



Tabular: ACI(D) Guarantee

Tabular: ACI(D) Guarantee

- Correctness
  - Achieve serializability the same way as OCC
  - Need some tweaks in commit protocol when `UpdateCallback()` is involved

```
def UpdateCallback(table, rid, cb):
  write_set.Add(table, rid, cb)
```

```
def Transaction::Commit(table, rid, callback):
  # Sort out-of-place and lock write set
  sorted_ws = sort(write_set)
  foreach w in sorted_ws:
    lock w

  ... fences and get commit epoch ...

  # Validate reads - exactly the same as OCC
  foreach r in read_set:
    if r.tid != r.record.get_tid() and r is not locked by me:
      rollback and return

  ... iterate read/write sets to get new commit_tid ...

  # Apply write callbacks and updates
  foreach w in write_set:
    if w.is_callback:
      w.callback() # invoke the callback
    else: # "normal" materialized update
      memcpy(w.record.data, w.data, w.size)
    w.tid = commit_tid

  ... other remaining operations ...
```

Tabular: ACI(D) Guarantee

- Correctness
  - Achieve serializability the same way as OCC
  - Need some tweaks in commit protocol when `UpdateCallback()` is involved

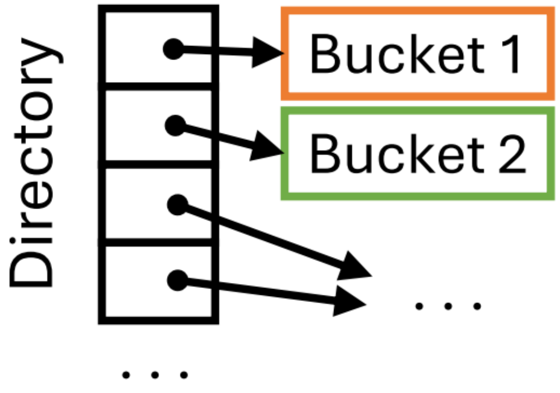
```
def UpdateCallback(table, rid, cb):
  write_set.Add(table, rid, cb)
```

- Transparent durability
  - Optionally via a per-table knob
  - Thread-local distributed logging and recovery

```
def Transaction::Commit(table, rid, callback):
  # Sort out-of-place and lock write set
  sorted_ws = sort(write_set)
  foreach w in sorted_ws:
    lock w

  ... fences and get commit epoch ...

  # Validate reads - exactly the same as OCC
  foreach r in read_set:
    if r.tid != r.record.get_tid() and r is not locked by me:
      rollback and return

  ... iterate read/write sets to get new commit_tid ...

  # Apply write callbacks and updates
  foreach w in write_set:
    if w.is_callback:
      w.callback() # invoke the callback
    else: # "normal" materialized update
      memcpy(w.record.data, w.data, w.size)
    w.tid = commit_tid

  ... other remaining operations ...
```

Case study: Tabular-based Hash-table

- Buckets are fixed-size records in a table
- Directory is stored in a separate table
- **Multiple tables as one data** structure is supported trivally



(a) Logical view

```
struct Bucket {
int n_keys;
  int local_depth;
  int lock;
  KVPair kvs[16];
};
```

(b) Bucket definition

```
struct Directory {
  int global_depth;
  Bucket *buckets[];
  int lock;
};
```

(c) Directory definition

RID	global_depth	buckets
0	0	…
1	1	…
2	2	…
3	3	…

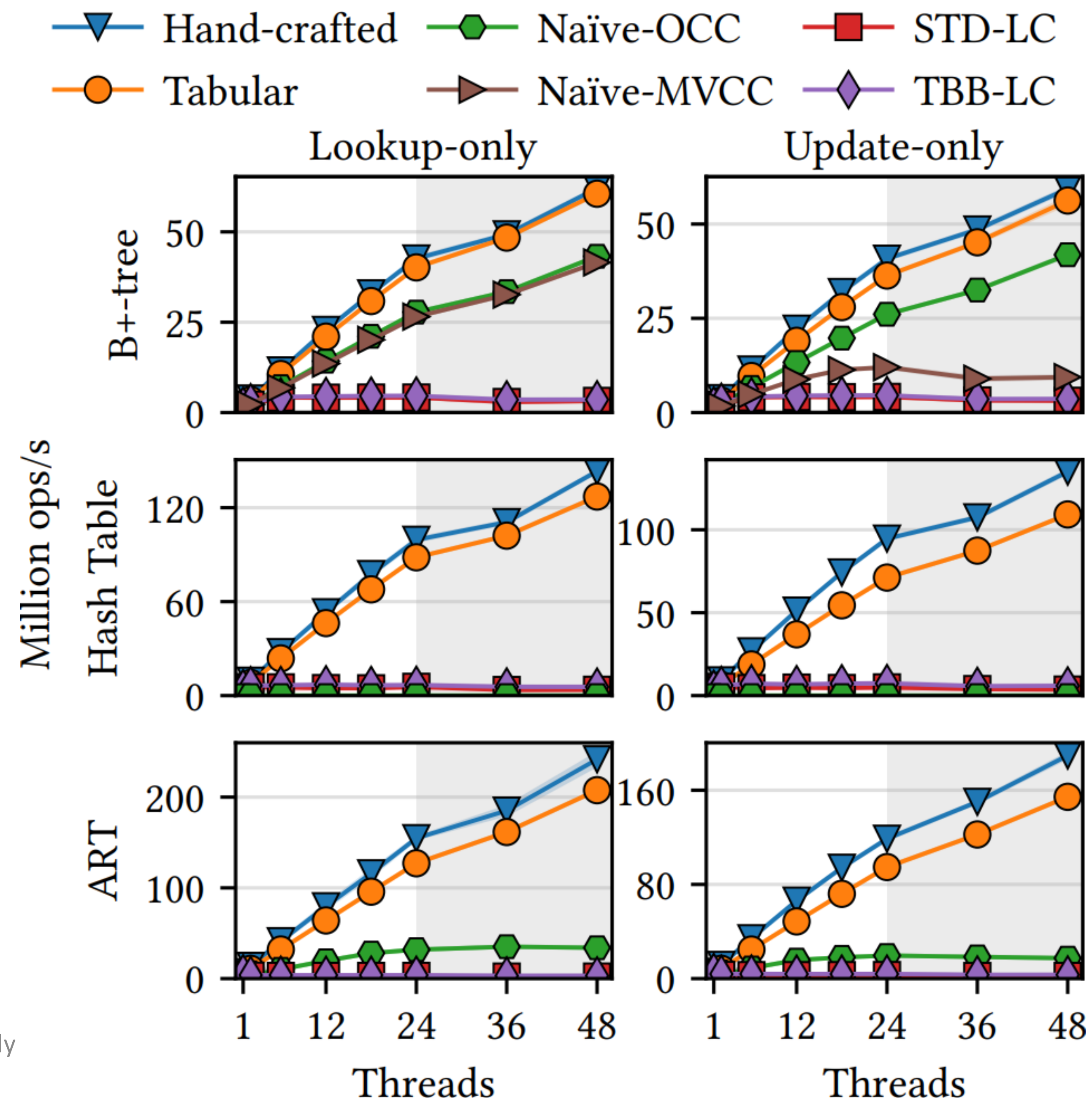
(d) Directory table

RID	n_keys	local_depth	kvs
0	10	2	...
1	8	2	...
2	4	3	...
...			

(e) Bucket table

Evaluation: YCSB-like index benchmarks

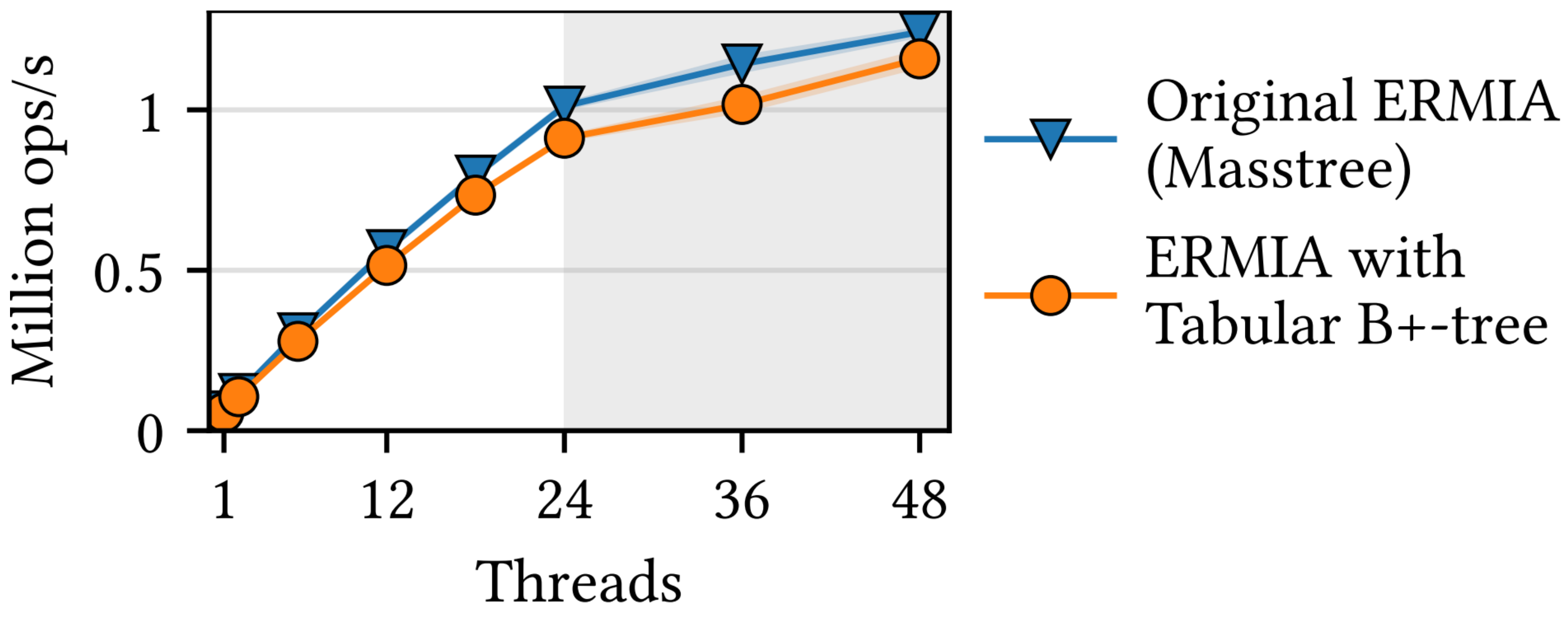
- 100M 8-byte keys and 8-byte values
- Index data structures
  - B+-tree
  - Hash-table (Extendible hashing)
  - Adaptive Radix Tree (ART)
- Variants
  - `Hand-crafted`, `Naïve-MVCC`, `Naïve-OCC`, `Tabular` are as aforementioned
  - `STD-LC`: Applying Pessimistic Lock Coupling using `std::shared_mutex`
  - `TBB-LC`: Same as above but using `tbb::spin_rw_mutex`



Tabular: Efficiently

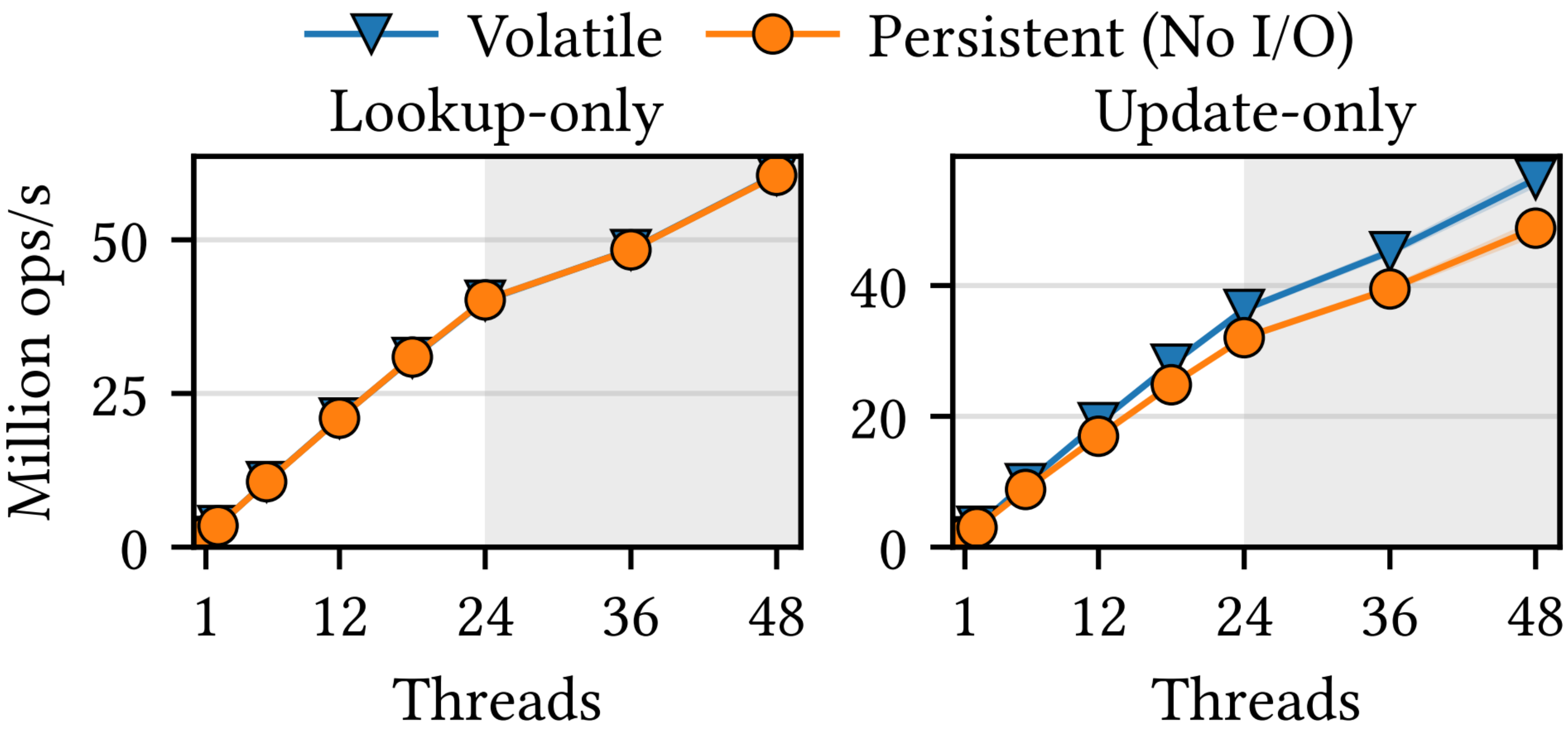
Evaluation: TPC-C

- ERMIA is a memory-resident OLTP engine
- Replace Masstree with Tabular-based B+-tree in ERMIA
- End-to-end performance is comparable to the original



Evaluation: Transparent Persistence

- In Persistent (No I/O), file operations are noops to show in-memory overhead
- Nearly zero overhead for lookups
- Less than 15% drop for updates



Summary

- Tabular is a parallel programming library with transactional interfaces
- Data structures (not only indexes) can be modelled as tables to be transparently concurrent and persistent
- Tabular-based indexes deliver competitive performance compared to their hand-crafted counterparts with drastically lower programming complexity

- Code is available at https://github.com/sfu-dis/tabular