Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes

Huanchen Zhang
David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, Rui Shen

PARALLEL DATA LABORATORY
Carnegie Mellon University
Part I
Initial Exploration of Hybrid Indexes
[SIGMOD’16]
You are running out of memory
You are running out of memory
You are running out of memory

Buy more

Yes

No
TPC-C on H-Store

Memory Limit = 5GB

Transactions Executed

Throughput

Memory (GB)

Disk tuples
In-memory tuples
Indexes
I GOT STUCK

SO I WENT TO SLEEP
The better way:

Use memory more efficiently
Indexes are **LARGE**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>% space for index</th>
<th>Hybrid Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPC-C</td>
<td>58%</td>
<td>34%</td>
</tr>
<tr>
<td>Voter</td>
<td>55%</td>
<td>41%</td>
</tr>
<tr>
<td>Articles</td>
<td>34%</td>
<td>18%</td>
</tr>
</tbody>
</table>
Our Contributions [SIGMOD’16]

1. The hybrid index architecture
2. The Dual-Stage Transformation
3. Applied to 4 index structures
   - B+tree
   - Masstree
   - Skip List
   - Adaptive Radix Tree (ART)

Performance ≈ Space
30 – 70%
Did we solve this problem?

Stay tuned

Transactions Executed
How do hybrid indexes achieve memory savings?
Hybrid Index: a dual-stage architecture

- dynamic stage
- static stage
Inserts are batched in the dynamic stage.
Reads search the stages in order

1. dynamic stage
2. static stage
A Bloom filter improves read performance

1. read
   dynamic stage

2. read
   static stage
Dynamic stage

Static stage

- Memory-efficient
- Skew-aware

1. Read
2. Merge

Read
Write
The Dual-Stage Transformation

1. Dynamic stage

2. Static stage

3. Merge
The Dual-Stage Transformation

1. dynamic stage
2. static stage

merge
The Dynamic-to-Static Rules

- Compaction
- Reduction
- Compression
The Dynamic-to-Static Rules

Compaction

Reduction

Compression
Compaction: minimize # of memory blocks
Compaction: minimize # of memory blocks
Reduction: minimize structural overhead
Reduction: minimize structural overhead
Reduction: minimize structural overhead

<p>| | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>h</td>
<td>i</td>
<td>j</td>
<td>k</td>
<td>l</td>
<td>m</td>
<td>n</td>
<td></td>
</tr>
</tbody>
</table>

Diagram showing a tree structure with nodes labeled from 1 to 12 and characters a to n.
The merge routine is a blocking process.
The merge routine is a blocking process.
Did we solve this problem?

Transactions Executed

Throughput (txn/s)

TPC-C on H-Store

B+tree
Yes, we improved the DBMS’s capacity!

TPC-C on H-Store

Throughput (txn/s)

Transactions Executed

B+tree

Hybrid
Transactions Executed Throughput (txn/s)

Memory (GB)

TPC-C on H-Store

B+tree

Disk tuples

Indexes

In-memory tuples

Hybrid

Transactions Executed
Transactions Executed

Throughput (txn/s)

Memory (GB)

TPC-C on H-Store

B+tree

Hybrid

Disk tuples

In-memory tuples

Indexes

Transactions Executed
<table>
<thead>
<tr>
<th>TPC-C on H-Store</th>
<th>Throughput (txn/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>B+tree</td>
</tr>
<tr>
<td></td>
<td>Hybrid</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory (GB)</th>
<th>Tuples</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>In-memory tuples</td>
</tr>
<tr>
<td>2M</td>
<td>Disk tuples</td>
</tr>
<tr>
<td>4M</td>
<td>Indexes</td>
</tr>
<tr>
<td>6M</td>
<td></td>
</tr>
<tr>
<td>8M</td>
<td></td>
</tr>
<tr>
<td>10M</td>
<td></td>
</tr>
</tbody>
</table>

Transactions Executed
Take Away:

- Memory saved by indexes
- Larger working set in memory
- Higher throughput

Transactions Executed

Memory (GB)

- B+tree
- Hybrid

Disk tuples
In-memory tuples
Indexes
Part I Recap

1. The hybrid index architecture

2. The Dual-Stage Transformation

3. Applied to 4 index structures
   - B+tree
   - Masstree
   - Skip List
   - Adaptive Radix Tree (ART)
Part II
Concurrent hybrid indexes with non-blocking merge
Building Concurrent Hybrid Index?

1. Read
   - Write
   - Dynamic stage

2. Read
   - Merge
   - Static stage
Building Concurrent Hybrid Index?

1. write
2. merge

dynamic stage

static stage
Use concurrent data structures for dynamic-stage

dynamic stage  static stage

1. read
2. read

write

merge
Static-stage is perfectly concurrent by default
Challenge: efficient non-blocking merge algorithm
Merge Algorithm Requirements

1. Non-blocking
   - All existing items are accessible during merge
   - New items can still enter

2. Efficient
   - Fast
   - Bounded temporary memory use
Naïve Solution 1: Coarse-grained Locking

1. Dynamic stage
2. Static stage

write
read
merge
Naïve Solution 1: Coarse-grained Locking

1. Read
2. Read

Write

Merge

Dynamic stage

Static stage

X
The intermediate stage unblocks write traffic

1. read
   - write

2. read
   - merge

dynamic stage  static stage
The intermediate stage unblocks write traffic

dynamic stage  Intermediate stage  static stage
The intermediate stage unblocks write traffic

dynamic stage  Intermediate stage  static stage
How do we unblock reads during merge?

Intermediate stage  static stage

2: read

3: read

merge
Naïve Solution 2: Full Copy-on-write

Intermediate stage

static stage
Key Observation

Merged-in items in the static-stage will NOT be accessed until the intermediate-stage is deleted.

Merge Incrementally!
Our Solution: Incremental Copy-on-write with Rapid GC
Our Solution: Incremental Copy-on-write with Rapid GC

When can we safely reclaim the garbage?
Our Solution: Incremental Copy-on-write with Rapid GC

When can we safely reclaim the garbage?
Our Solution: Incremental Copy-on-write with Rapid GC

When no thread still holds a reference to it!
Our Solution: Incremental Copy-on-write with Rapid GC

**Thread-local counters**

When no thread still holds a reference to it!
Our Solution: Incremental Copy-on-write with Rapid GC

Thread-local counters

\[ ++C_i = \text{MAX}(C_i, C_{\text{max}}) + 1 \]

GC Condition:

\[ C_{\text{min}} > \text{garbage tag} \]

Our Solution:

Incremental Copy-on-write with Rapid GC
A Quick Recap of the Merge Algorithm

1. The **intermediate stage** separates writes from the merge process.

2. The **incremental merge** algorithm with **rapid GC** is non-blocking and space-efficient.
What we are building now

Non-blocking Merge Compact Radix Tree
What we are building now

Non-blocking Merge

Compact Radix Tree
What we are building now

Non-blocking Merge

Compact Radix Tree
What we are building now

Bwtree

Non-blocking Merge

Compact Radix Tree
What we are building now

Skiplist

Non-blocking Merge

Compact Radix Tree
What we are building now

Masstree

Non-blocking Merge

Compact Radix Tree
Part III
Super-compact static-stage
Go “crazy” on space-efficiency

Succinct Data Structures
- $Z + o(Z)$, where $Z$ is the information-theoretic lower bound
- Still allow for efficient query operations

100011010000101…

(rank\(_1\)(x) = \# of 1’s up to position x

select\(_1\)(x) = \text{position of the } x\text{-occurrence of 1}
Encoding Radix Tree

- **a**
  - $ab$
    - $lnr$
      - $iio$
        - $i$
          - i$n$
            - $$
  - $a$

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Binary1</th>
<th>Binary2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>$ab$</td>
<td>100</td>
<td>1000</td>
</tr>
<tr>
<td>$lnr$a</td>
<td>100010</td>
<td>10000100</td>
</tr>
<tr>
<td>$iiio$</td>
<td>10001</td>
<td>100101010</td>
</tr>
<tr>
<td>i$n$</td>
<td>010</td>
<td>101010</td>
</tr>
<tr>
<td>$#$</td>
<td>11</td>
<td>1010</td>
</tr>
</tbody>
</table>
Memory Savings with the New Encoding

- ART: 1000 MB
- Our Encoding: 166 MB

84% Memory Savings

50M email keys with average length = 20 bytes
The Takeaway Message

Hybrid indexes can save the precious memory resources with minimum performance penalty.
Toll-Free Hotline:

1-844-88-CMUDB
Back-up Slides
<table>
<thead>
<tr>
<th></th>
<th>B+tree</th>
<th>Hybrid</th>
</tr>
</thead>
<tbody>
<tr>
<td>50%</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>99%</td>
<td>50</td>
<td>52</td>
</tr>
<tr>
<td>MAX</td>
<td>115</td>
<td>611</td>
</tr>
</tbody>
</table>
YCSB-based Microbenchmark Evaluation

**Workload:** insert, read/update (50/50)

**Key:** email

**Value:** 64-bit unsigned integer (pointer)

† Single thread

**50M** entries, **10M** queries (Zipf distributed)
Hybrid index saves memory

Memory (GB)

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>Hybrid</th>
</tr>
</thead>
<tbody>
<tr>
<td>B+tree</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Masstree</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Skip List</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ART</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

30 – 70%
Hybrid index provides comparable throughput

**Read/Update (50/50)**

- B+tree: 4M
- Masstree: 2M
- Skip List: 0
- ART: 16M

**Insert-only**

- B+tree: 4M
- Masstree: 2M
- Skip List: 0
- ART: 4M