ADMINISTRIVIA

Peloton **master branch** has been updated to provide easier to use debug methods.
→ Your implementation should match the behavior of the Bw-Tree.

We will be sending out information on how to access the MemSQL development machines.
TODAY’S AGENDA

T-Tree
Skip List
Bw-Tree
T-TREES

Based on AVL Trees. Instead of storing keys in nodes, store pointers to their original values.

Proposed in 1986 from Univ. of Wisconsin Used in TimesTen and other early in-memory DBMSs during the 1990s.
T-Trees

T-Tree Node

- Parent Pointer
- Data Pointers
- Min-K
- Max-K
- Left Child Pointer
- Right Child Pointer
T-TREES

T-Tree Node

Node Boundaries

Parent Pointer

Min-K

Max-K

Left Child Pointer

Right Child Pointer
**T-TREES**

**Key Space (Low→High)**

1 2 3 4 5 6 7
**T-TREES**

**Advantages**
→ Uses less memory because it does not store keys inside of each node.
→ Inner nodes contain key/value pairs (like B-Tree).

**Disadvantages**
→ Difficult to rebalance.
→ Difficult to implement safe concurrent access.
→ Have to chase pointers when scanning range or performing binary search inside of a node.
OBSERVATION

The easiest way to implement a **dynamic** order-preserving index is to use a sorted linked list. All operations have to linear search. → Average Cost: $O(N)$
OBSERVATION

The easiest way to implement a dynamic order-preserving index is to use a sorted linked list. All operations have to linear search. → Average Cost: O(N)
The easiest way to implement a \textbf{dynamic} order-preserving index is to use a sorted linked list.
All operations have to linear search.
→ Average Cost: $O(N)$
SKIP LISTS

Multiple levels of linked lists with extra pointers that skip over intermediate nodes.

Maintains keys in sorted order without requiring global rebalancing.
SKIP LISTS

A collection of lists at different levels
→ Lowest level is a sorted, singly linked list of all keys
→ 2nd level links every other key
→ 3rd level links every fourth key
→ In general, a level has half the keys of one below it

To insert a new key, flip a coin to decide how many levels to add the new key into.
Provides approximate $O(\log n)$ search times.
SKIP LISTS: EXAMPLE

Levels

- $P = N/4$
- $P = N/2$
- $P = N$

End

- $\infty$

K1
V1

K2
V2

K3
V3

K4
V4

K6
V6
**SKIP LISTS: EXAMPLE**

![Diagram of skip lists with levels and nodes labeled.](image)
**SKIP LISTS: EXAMPLE**

**Levels**

- **$P = \frac{N}{4}$**
- **$P = \frac{N}{2}$**
- **$P = N$**

**End**

- **$\infty$**
- **$\infty$**
- **$\infty$**
SKIP LISTS: INSERT

Insert K5

Levels

- $P=N$
- $P=N/2$
- $P=N/4$

End

- $\infty$
- $\infty$
- $\infty$

Node insertions:
- Insert K5

Node relationships:
- K1 to V1
- K2 to V2
- K3 to V3
- K4 to V4
- K6 to V6
**SKIP LISTS: INSERT**

*Insert K5*

- **Levels**
  - $P=N/4$
  - $P=N/2$
  - $P=N$

- **End**
  - $\infty$

**Diagram:**
- Nodes: $K1, V1, K2, V2, K3, V3, K4, V4, K5, V5, K6, V6, K5$
- Paths:
  - $K1 \rightarrow K2 \rightarrow K3 \rightarrow K4 \rightarrow K5 \rightarrow K6$
  - $V1 \rightarrow V2 \rightarrow V3 \rightarrow V4 \rightarrow V5 \rightarrow V6$

Notes:
- $P=N$ indicates the total number of elements.
- $P=N/2$ and $P=N/4$ likely represent different levels or probabilities in skip list insertion.
- The diagram illustrates the insertion process for $K5$, showing the traversal through levels 1 to 4.
SKIP LISTS: INSERT

Insert K5

Levels

- $P = N/4$
- $P = N/2$
- $P = N$

- $K1, V1$
- $K2, V2$
- $K3, V3$
- $K4, V4$
- $K5, V5$
- $K6, V6$

End

- $\infty$
- $\infty$
- $\infty$
Skip Lists: Insert

Insert $K_5$

Levels

$P = N$

$K_1 \rightarrow K_2 \rightarrow K_3 \rightarrow K_4 \rightarrow K_5 \rightarrow K_6 \rightarrow \infty$

$P = N/4$

$K_1 \rightarrow K_2 \rightarrow K_3 \rightarrow K_4 \rightarrow K_5 \rightarrow K_6 \rightarrow \infty$

$P = N/2$

$K_1 \rightarrow K_2 \rightarrow K_3 \rightarrow K_4 \rightarrow K_5 \rightarrow K_6 \rightarrow \infty$
**SKIP LISTS: SEARCH**

*Find K3*

Levels

- **P=N**: K1 V1
- **P=N/2**: K2 V2
- **P=N/4**: K3 V3

End

- **∞**: K5
- **∞**: K6

Diagram shows the process of finding key K3 in a skip list.
**SKIP LISTS: SEARCH**

**Find K3**

- **Levels**
  - $P = N/4$
  - $P = N/2$
  - $P = N$

- **End**
  - $\infty$

K3 < K5

- K3 < K5
- K1 < K2
- K2 < K3
- K3 < K4
- K4 < K5
- K5 < K6
SKIP LISTS: SEARCH

Find K3

Levels

P=N
K1
V1
K2
V2
K3
V3
K4
V4
K5
V5
K6
V6

End

∞

∞

∞

K3<K5
K3>K2

P=N

P=N/2

P=N/4

Find K3
K3<K5
K3>K2
SKIP LISTS: SEARCH

Find $K_3$

- $K_3 < K_5$
- $K_3 > K_2$
- $K_3 < K_4$

Levels:
- $P = N$
- $P = N/2$
- $P = N/4$

End:
- $\infty$
SKIP LISTS: SEARCH

Find K3

Levels

K3<K5
P=N/4

K3<K4
K3<K5
P=N/2

K3>K2
P=N

K3<K4

End

K5
∞

K6
∞

K5
∞

K5
∞

K5
∞
SKIP LISTS: ADVANTAGES

Uses less memory than a typical B+tree (only if you don’t include reverse pointers).

Insertions and deletions do not require rebalancing.

It is possible to implement a concurrent skip list using only CAS instructions.
CONCURRENT SKIP LIST

Can implement insert and delete without locks using only CaS operations.

The data structure only support links in one direction because CaS can only swap one pointer atomically.
**SKIP LISTS: INSERT**

**Insert K5**

```
Levels

P=N/4
K1 P=N/2 K2 K3 P=N
V1 K2 V2 K3 V4
V1 K2 V2 K3 V4
K5

End

∞ ∞ ∞ K6 V6
```

- **P=N/4**
- **P=N/2**
- **P=N**

The diagram shows the process of inserting key K5 into a skip list structure.
SKIP LISTS: INSERT

Insert K5

Levels

$\text{Level 1: } P=N$  
$\text{Level 2: } P=N/2$  
$\text{Level 3: } P=N/4$  

End

$K1$  
$V1$  

$K2$  
$V2$  

$K3$  
$V3$  

$K4$  
$V4$  

$K5$  
$V5$  

$K6$  
$V6$  

$\infty$
**SKIP LISTS: INSERT**

*Insert K5*

- Levels: `K1 V1`  `K2 V2`  `K3 V3`  `K4 V4`  `K5 V5`  `K6 V6`
- End: `∞`  `∞`  `∞`

**Levels**
- `P=N`
- `P=N/2`  `P=N/4`

**Insert K5**
- `K1 V1`  `K2 V2`  `K3 V3`  `K4 V4`  `K5 V5`  `K6 V6`
**SKIP LISTS: INSERT**

**Insert K5**

Levels

- $P = N/4$
- $P = N/2$
- $P = N$

End

- $\infty$

Diagram: Inserting $K5$ into skip lists with levels $P = N$, $P = N/2$, and $P = N/4$.
**SKIP LISTS: INSERT**

**Insert K5**

Levels

- $P=N$ (Level 1)
- $P=N/2$ (Level 2)
- $P=N/4$ (Level 3)

End

- $\infty$ (Level 1)
- $\infty$ (Level 2)
- $\infty$ (Level 3)

**Insertion Process**

1. **Level 1**
   - Insert $K1$ with value $V1$
   - Insert $K2$ with value $V2$
   - Insert $K3$ with value $V3$
   - Insert $K4$ with value $V4$

2. **Level 2**
   - Insertion of $K5$ at Level 2
   - CaS (Compare and Swap) occurs at Level 2
   - Insertion of $K6$ with value $V6$

3. **Level 3**
   - Insertion of $K7$ with value $V7$

The diagram illustrates the process of inserting $K5$ into the skip list, highlighting the levels and the values inserted at each level.
SKIP LISTS: INSERT

Insert K5

Levels

End

$P=N$

$K1$ $V1$

$K2$ $V2$

$K3$ $V3$

$K4$ $V4$

$K5$ $V5$

$K6$ $V6$

$P=N/4$

$P=N/2$

CaS

CaS

$∞$

$∞$

$∞$
**Insert K5**

Levels:
- **CaS**
  - $P = N/4$
  - Insert $K5$
  - CaS
- $P = N/2$
  - $K1$
  - $V1$
  - $K2$
  - $V2$
  - $K3$
  - $V3$
  - $K4$
  - $V4$
  - CaS
- $P = N$

End:
- $\infty$
- Insert $K5$
- $\infty$
- $\infty$
First *logically* remove a key from the index by setting a flag to tell threads to ignore.

Then *physically* remove the key once we know that no other thread is holding the reference.

→ Perform CaS to update the predecessor’s pointer.
Skip Lists: Delete

Delete K5

Levels

- P=N
- P=N/2
- P=N/4

End

- K1
- K2
- K3
- K4
- K5
- K6

V1
V2
V3
V4
V5
V6

Del false
Del false
Del false
Del false
Del false
Del false

∞
∞
∞
∞
∞
SKIP LISTS: DELETE

Delete K5

Levels

End

P=N

P=N/4

K1 V1 Del false

K2 V2 Del false

K3 V3 Del false

K4 V4 Del false

K5 V5 Del true

K6 V6 Del false

P=N/2

K2

K4

K5

K6

∞

∞

∞

Del false

Del false

Del false

Del false

Del true

Del false
Skip Lists: Delete

Delete K5

Levels

P=N

K1  V1  Del false

P=N/2

K2  V2  Del false

P=N/4

K3  V3  Del false

K4  V4  Del false

K5  V5  Del true

K6  V6  Del false

End

∞

∞

∞
SKIP LISTS: DELETE

Delete K5

Levels

End

P=N

K1 V1 Del false

K2 V2 Del false

K3 V3 Del false

K4 V4 Del false

K5 V5 Del true

K6 V6 Del false

P=N/4

P=N/2

K2

K4

K5

K6

K5

K5

K5

∞

∞

∞
SKIP LISTS: DELETE

Delete K5

Levels

P=N/4

K1

V1

Del false

P=N/2

K2

V2

Del false

P=N

K3

V3

Del false

K4

V4

Del false

K5

V5

Del true

K6

V6

Del false

End

∞
CONCURRENT SKIP LIST

Be careful about how you order operations.

If the DBMS invokes operation on the index, it can never “fail”
→ A txn can only abort due to higher-level conflicts.
→ If a CaS fails, then the index will retry until it succeeds.
SKIP LIST OPTIMIZATIONS

Reducing `RAND()` invocations.

Packing multiple keys in a node.

Reverse iteration with a stack.

Reusing nodes with memory pools.
**SKIP LIST: COMBINE NODES**

Store multiple keys in a single node.

→ **Insert Key**: Find the node where it should go and look for a free slot. Perform CaS to store new key. If no slot is available, insert new node.

→ **Search Key**: Perform linear search on keys in each node.

![Node Diagram]

Source: Ticki

CMU 15-721 (Spring 2018)
SKIP LIST: COMBINE NODES

Store multiple keys in a single node.

→ **Insert Key:** Find the node where it should go and look for a free slot. Perform CaS to store new key. If no slot is available, insert new node.

→ **Search Key:** Perform linear search on keys in each node.
**SKIP LIST: COMBINE NODES**

Store multiple keys in a single node.

→ **Insert Key:** Find the node where it should go and look for a free slot. Perform CaS to store new key. If no slot is available, insert new node.

→ **Search Key:** Perform linear search on keys in each node.

*Insert K4*

```
  K2  K3  K6  -  
  V2  V3  V6  -  
```
SKIP LIST: COMBINE NODES

Store multiple keys in a single node.

→ **Insert Key:** Find the node where it should go and look for a free slot. Perform CaS to store new key. If no slot is available, insert new node.

→ **Search Key:** Perform linear search on keys in each node.

**Insert K4**

```
K2  K3  K6  K4
V2  V3  V6  V4
```

Source: Ticki
**SKIP LIST: COMBINE NODES**

Store multiple keys in a single node.

→ **Insert Key**: Find the node where it should go and look for a free slot. Perform CaS to store new key. If no slot is available, insert new node.

→ **Search Key**: Perform linear search on keys in each node.

**Search K6**

```
K2  K3  K6  K4
V2  V3  V6  V4
```
SKIP LISTS: REVERSE SEARCH

Find \([K4, K2]\)

Levels

\(K2 < K5\)

\(P = N/4\)

\(P = N/2\)

\(P = N\)

End

Source: Mark Papadakis

CMU 15-721 (Spring 2018)
**SKIP LISTS: REVERSE SEARCH**

Find \([K4,K2]\)

**Levels**

- **P=N**: All elements
- **P=N/2**: Elements between 1 and N/2
- **P=N/4**: Elements between N/4 and 3N/4

**End**

- K5
- K6
- K7

**K2 < K5**

- K2
- K5

**K2 = K2**

- K2
- K2

Source: Mark Papadakis
SKIP LISTS: REVERSE SEARCH

Find \([K4,K2]\)

Levels

- \(P=N\)
- \(K2=K2\)
- \(K2<K5\)

End

\(\infty\)

Source: Mark Papadakis
**SKIP LISTS: REVERSE SEARCH**

Find \([K4,K2]\)

Stack:

Levels

- **K2<K5**
  - \(P=N/4\)
  - \(K2< K5\)

- **K2=K2**
  - \(P=N/2\)
  - \(K2=K2\)

- **K2<K5**
  - \(P=N\)
  - \(K1\)
  - \(V1\)

- **K2<K5**
  - \(K2\)
  - \(V2\)

- **K2<K5**
  - \(K3\)
  - \(V3\)

- **K2<K5**
  - \(K4\)
  - \(V4\)

- **K2<K5**
  - \(K5\)
  - \(V5\)

- **K2<K5**
  - \(K6\)
  - \(V6\)

End

Source: Mark Papadakis

CMU 15-721 (Spring 2018)
SKIP LISTS: REVERSE SEARCH

Find \([K4,K2]\)

Stack:

\[ K3 \quad K2 \]

End

Source: Mark Papadakis
SKIP LISTS: REVERSE SEARCH

Find \([K4, K2]\)

Stack:

Source: Mark Papadakis
SKIP LISTS: REVERSE SEARCH

Find \([K4, K2]\)

Stack: \([K4, K3, K2]\)

Source: Mark Papadakis
SKIP LISTS: REVERSE SEARCH

Find [K4,K2]

Stack:

End

K2<K5

K2=K2

Source: Mark Papadakis
OBSERVATION

Because CaS only updates a single address at a time, this limits the design of our data structures.

- We cannot have reverse pointers in a latch-free concurrent Skip List.
- We cannot build a latch-free B+Tree.

What if we had an indirection layer that allowed us to update multiple addresses atomically?
Latch-free B+Tree index
→ Threads never need to set latches or block.

**Key Idea #1: Deltas**
→ No updates in place
→ Reduces cache invalidation.

**Key Idea #2: Mapping Table**
→ Allows for CaS of physical locations of pages.
BW-TREE: MAPPING TABLE

Mapping Table

<table>
<thead>
<tr>
<th>PID</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td></td>
</tr>
</tbody>
</table>

Index Page

101

102

104

Logical Pointer

Physical Pointer
## BW-TREE: MAPPING TABLE

### Mapping Table

<table>
<thead>
<tr>
<th>PID</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td></td>
</tr>
</tbody>
</table>

### Index Page

Logical Pointer

Physical Pointer

Index Page

102

104

104

102
BW-TREE: DELTA UPDATES

Mapping Table

<table>
<thead>
<tr>
<th>PID</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td></td>
</tr>
</tbody>
</table>

Each update to a page produces a new delta.
Each update to a page produces a new delta.

Delta physically points to base page.

Source: Justin Levandoski
Each update to a page produces a new delta.

Delta physically points to base page.

Install delta address in physical address slot of mapping table using CAS.

Source: Justin Levandoski
Each update to a page produces a new delta.

Delta physically points to base page.

Install delta address in physical address slot of mapping table using CAS.
Each update to a page produces a new delta.

Delta physically points to base page.

Install delta address in physical address slot of mapping table using CAS.

Source: Justin Levandoski
Each update to a page produces a new delta.

Delta physically points to base page.

Install delta address in physical address slot of mapping table using CAS.

Source: Justin Levandoski

CMU 15-721 (Spring 2018)
BW-TREE: SEARCH

**Mapping Table**

<table>
<thead>
<tr>
<th>PID</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td></td>
</tr>
</tbody>
</table>

Traverse tree like a regular B+tree.

If mapping table points to delta chain, stop at first occurrence of search key.
BW-TREE: SEARCH

Traverse tree like a regular B+tree.

If mapping table points to delta chain, stop at first occurrence of search key.

Otherwise, perform binary search on base page.
BW-TREE: CONTENTION UPDATES

Mapping Table

<table>
<thead>
<tr>
<th>PID</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td></td>
</tr>
</tbody>
</table>

Threads may try to install updates to the same state of the page.

Logical Pointer

Physical Pointer

Page 102

▲ Insert 50

Page 102
Threads may try to install updates to the same state of the page. Winner succeeds, any losers must retry or abort.
Threads may try to install updates to the same state of the page.

Winner succeeds, any losers must retry or abort.
BW-TREE: DELTA TYPES

Record Update Deltas
→ Insert/Delete/Update of record on a page

Structure Modification Deltas
→ Split/Merge information
Consolidate updates by creating new page with deltas applied.
Consolidate updates by creating new page with deltas applied.

CAS-ing the mapping table address ensures no deltas are missed.

**Mapping Table**

<table>
<thead>
<tr>
<th>PID</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td>102</td>
</tr>
<tr>
<td>103</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td></td>
</tr>
</tbody>
</table>
**BW-TREE: CONSOLIDATION**

Consolidate updates by creating a new page with deltas applied.

CAS-ing the mapping table address ensures no deltas are missed.

### Mapping Table

<table>
<thead>
<tr>
<th>PID</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td></td>
</tr>
</tbody>
</table>

- **Insert 55**
- **Delete 48**
- **Insert 50**

Page 102

New 102
Consolidate updates by creating a new page with deltas applied.

CAS-ing the mapping table address ensures no deltas are missed.

Old page + deltas are marked as garbage.
Operations are tagged with an **epoch**
→ Each epoch tracks the threads that are part of it and the objects that can be reclaimed.
→ Thread joins an epoch prior to each operation and post objects that can be reclaimed for the current epoch (not necessarily the one it joined)

Garbage for an epoch reclaimed only when all threads have exited the epoch.
BW-TREE: GARBAGE COLLECTION

Mapping Table

<table>
<thead>
<tr>
<th>PID</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td></td>
</tr>
</tbody>
</table>

Logical Pointer

Physical Pointer

Epoch Table

Page 102

New 102

Insert 55

Delete 48

Insert 50

CMU 15-721 (Spring 2018)
**BW-TREE: GARBAGE COLLECTION**

**Mapping Table**

<table>
<thead>
<tr>
<th>PID</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td></td>
</tr>
</tbody>
</table>

**Epoch Table**

- **Insert 55**
- **Delete 48**
- **Insert 50**

**Logical Pointer**

**Physical Pointer**

**New 102**

**Page 102**

**CPU1**
BW-TREE: GARBAGE COLLECTION

Mapping Table

<table>
<thead>
<tr>
<th>PID</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td></td>
</tr>
</tbody>
</table>

Logical Pointer

Physical Pointer

Epoch Table

CPU2

CPU1

Insert 55

Delete 48

Insert 50

Page 102

New 102
BW-TREE: GARBAGE COLLECTION

Mapping Table

<table>
<thead>
<tr>
<th>PID</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td></td>
</tr>
</tbody>
</table>

Epoch Table

CPU2
- Insert 55
- Delete 48
- Insert 50

CPU1
- Insert 55
- Delete 48
- Insert 50

Page 102

Logical Pointer

Physical Pointer

New 102

CPU2

CPU1
**BW-TREE: GARBAGE COLLECTION**

**Mapping Table**

<table>
<thead>
<tr>
<th>PID</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td></td>
</tr>
</tbody>
</table>

**Logical Pointer**

**Physical Pointer**

**Insert 55**

**Delete 48**

**Insert 50**

**Page 102**

**New 102**

**Epoch Table**

**CPU2**

**Insert 55**

**Delete 48**

**Insert 50**

**Page 102**
**BW-TREE: GARBAGE COLLECTION**

### Mapping Table

<table>
<thead>
<tr>
<th>PID</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td></td>
</tr>
</tbody>
</table>

### Epoch Table

- **Insert 55**
- **Delete 48**
- **Insert 50**

Page 102

**Logical Pointer**

**Physical Pointer**

New 102
**BW-TREE: GARBAGE COLLECTION**

**Mapping Table**

<table>
<thead>
<tr>
<th>PID</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td></td>
</tr>
</tbody>
</table>

**Logical Pointer**

**Physical Pointer**

**Epoch Table**

- Insert 55
- Delete 48
- Insert 50

Page 102

New 102
BW-TREE: STRUCTURE MODIFICATIONS

Split Delta Record
→ Mark that a subset of the base page’s key range is now located at another page.
→ Use a logical pointer to the new page.

Separator Delta Record
→ Provide a shortcut in the modified page’s parent on what ranges to find the new page.
**BW-TREE: STRUCTURE MODIFICATIONS**

**Mapping Table**

<table>
<thead>
<tr>
<th>PID</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td></td>
</tr>
</tbody>
</table>

**Logical Pointer**

**Physical Pointer**

Diagram showing nodes labeled 1 to 8 with pointers indicating structure modifications.
BW-TREE: STRUCTURE MODIFICATIONS

Mapping Table

<table>
<thead>
<tr>
<th>PID</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td></td>
</tr>
</tbody>
</table>

Logical Pointer

Physical Pointer

Diagram shows the structure of a BW-tree with logical and physical pointers. The diagram includes a mapping table with PID and Addr columns, and a graphical representation of the tree structure with nodes labeled 1 to 8 and pointers connecting them.
# BW-TREE: STRUCTURE MODIFICATIONS

## Mapping Table

<table>
<thead>
<tr>
<th>PID</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td></td>
</tr>
</tbody>
</table>

---

### Logical Pointer

- 1 → 2
- 3 → 4 → 5 → 6
- 7 → 8

### Physical Pointer

- 101
- 102
- 103
- 104
- 105
**BW-TREE: STRUCTURE MODIFICATIONS**

**Mapping Table**

<table>
<thead>
<tr>
<th>PID</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td></td>
</tr>
</tbody>
</table>

- **Logical Pointer**
- **Physical Pointer**
BW-TREE: STRUCTURE MODIFICATIONS

Mapping Table

<table>
<thead>
<tr>
<th>PID</th>
<th>Addr</th>
<th>Logical Pointer</th>
<th>Physical Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>103</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>104</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>105</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 101
- 102
- 103
- 104
- 105

Split

105

102

103

104

101
BW-TREE: STRUCTURE MODIFICATIONS

Mapping Table

<table>
<thead>
<tr>
<th>PID</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td></td>
</tr>
</tbody>
</table>

Logical Pointer

Physical Pointer

▲ Split
**BW-TREE: STRUCTURE MODIFICATIONS**

**Mapping Table**

<table>
<thead>
<tr>
<th>PID</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td></td>
</tr>
</tbody>
</table>

**Diagram Description**

- **Logical Pointer**: Points to the nodes in the tree structure.
- **Physical Pointer**: Connects the logical pointers to their physical addresses in the memory.

The diagram illustrates a split operation in the BW-tree, where node 3 and 4 are split into two new nodes, 3 and 4, while node 5 and 6 are merged into a single node. The split is indicated by the red triangle labeled "Split."
BW-TREE: STRUCTURE MODIFICATIONS

Mapping Table

<table>
<thead>
<tr>
<th>PID</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td></td>
</tr>
</tbody>
</table>
**BW-TREE: STRUCTURE MODIFICATIONS**

**Mapping Table**

<table>
<thead>
<tr>
<th>PID</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td></td>
</tr>
</tbody>
</table>

**Logical Pointer**

**Physical Pointer**

- Split indicator
- Logical Pointer connections
- Physical Pointer connections
**BW-TREE: STRUCTURE MODIFICATIONS**

Mapping Table

<table>
<thead>
<tr>
<th>PID</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td></td>
</tr>
</tbody>
</table>

**Logical Pointer**

**Physical Pointer**

Split
**BW-TREE: STRUCTURE MODIFICATIONS**

**Mapping Table**

<table>
<thead>
<tr>
<th>PID</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td></td>
</tr>
</tbody>
</table>

**Logical Pointer**

**Physical Pointer**
BW-TREE: STRUCTURE MODIFICATIONS

Mapping Table

<table>
<thead>
<tr>
<th>PID</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td></td>
</tr>
</tbody>
</table>

- **Separator** at 101
- **Split** at 103

Logical Pointer
Physical Pointer
BW-TREE: STRUCTURE MODIFICATIONS

Mapping Table

<table>
<thead>
<tr>
<th>PID</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td></td>
</tr>
</tbody>
</table>

- **Separator** at PID 101
- **Split** at PID 103

Logical Pointer

Physical Pointer
BW-TREE: STRUCTURE MODIFICATIONS

Mapping Table

<table>
<thead>
<tr>
<th>PID</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td></td>
</tr>
</tbody>
</table>

Separator

Split

(Logical Pointer) [\(-\infty,3) [3,7) [7,\infty) ([5,7)]

Physical Pointer
BW-TREE: STRUCTURE MODIFICATIONS

Mapping Table

<table>
<thead>
<tr>
<th>PID</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td></td>
</tr>
</tbody>
</table>

Logical Pointer

Physical Pointer

Separator

Split

[1, 2)
[3, 4)
[5, 7)
[7, ∞)

[-∞, 3) [3, 7) [7, ∞)
**BW-TREE: PERFORMANCE**

Processor: 1 socket, 4 cores w/ 2×HT

![Bar chart showing performance comparison between Bw-Tree, B+Tree, and Skip List for Xbox, Synthetic, and Deduplication operations.](chart.png)

- **Bw-Tree** operations/sec (M): 10.4, 3.83, 2.84
- **B+Tree** operations/sec (M): 0.56, 0.66, 0.33
- **Skip List** operations/sec (M): 4.23, 1.02, 0.72

Source: Justin Levandoski
BW-TREE: PERFORMANCE

Processor: 1 socket, 10 cores w/ 2×HT
Workload: 50m Random Integer Keys (64-bit)

<table>
<thead>
<tr>
<th>Operations/sec (M)</th>
<th>Open Bw-Tree</th>
<th>B+Tree</th>
<th>Skip List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert-Only</td>
<td>9.94</td>
<td>8.09</td>
<td>2.51</td>
</tr>
<tr>
<td>Read-Only</td>
<td>15.5</td>
<td>29</td>
<td>2.78</td>
</tr>
<tr>
<td>Read/Update</td>
<td>13.3</td>
<td>1.51</td>
<td>25.1</td>
</tr>
</tbody>
</table>

Source: Ziqi Wang
BW-TREE: PERFORMANCE

Processor: 1 socket, 10 cores w/ 2xHT
Workload: 50m Random Integer Keys (64-bit)

- Open Bw-Tree
- B+Tree
- Skip List
- Masstree
- ART

Source: Ziqi Wang

Insert-Only
- Operations/sec (M)
- Open Bw-Tree: 9.94
- B+Tree: 8.09
- Skip List: 17.9
- Masstree: 44.9
- ART: 2.51

Read-Only
- Operations/sec (M)
- Open Bw-Tree: 15.5
- B+Tree: 29
- Skip List: 2.78
- Masstree: 30.5
- ART: 1.51

Read/Update
- Operations/sec (M)
- Open Bw-Tree: 13.3
- B+Tree: 25.1
- Skip List: 22
- Masstree: 42.9
- ART: 44.9
PARTING THOUGHTS

Managing a concurrent index looks a lot like managing a database.

Non-concurrent Skip List is easy to implement. A Bw-Tree is hard to implement.
NEXT CLASS

Let's add latches back in our OLTP indexes!
Other implementation issues.
Crash course on performance testing.