TODAY’S AGENDA

Order Preserving Indexes
Index Locking & Latching
Prison Gang Tattoos
DATABASE INDEX

A data structure that improves the speed of data retrieval operations on a table at the cost of additional writes and storage space. Indexes are used to quickly locate data without having to search every row in a table every time a table is accessed.
DATA STRUCTURES

Order Preserving Indexes
→ A tree-like structure that maintains keys in some sorted order.
→ Supports all possible predicates with $O(\log n)$ searches.

Hashing Indexes
→ An associative array that maps a hash of the key to a particular record.
→ Only supports equality predicates with $O(1)$ searches.
Order Preserving Indexes

→ A tree-like structure that maintains keys in some sorted order.
→ Supports all possible predicates with $O(\log n)$ searches.

Hashing Indexes

→ An associative array that maps a hash of the key to a particular record.
→ Only supports equality predicates with $O(1)$ searches.
B-TREE VS. B+TREE

The original B-tree from 1972 stored keys + values in all nodes in the tree. → More memory efficient since each key only appears once in the tree.

A B+tree only stores values in leaf nodes. Inner nodes only guide the search process. → Easier to manage concurrent index access when the values are only in the leaf nodes.
B+ TREE
B+TREE

B+Tree Leaf Node

Prev K1 V1 • • • Kn Vn Next
B+TREE

B+Tree Leaf Node

Prev  K1  V1  ...  Kn  Vn  Next
B+TREE

B+Tree Leaf Node

Key + Value

Prev

$K_1, V_1$

$K_n, V_n$

Next
B+TREE

B+Tree Leaf Node

Key + Value

Prev

Next
B+Tree Leaf Node

- Sorted Keys: K1, K2, K3, K4, K5, ..., Kn
- Values: [values]

- Level: #
- Slots: #
- Prev: ★
- Next: ★

B+TREE
B+TREE

B+Tree Leaf Node

<table>
<thead>
<tr>
<th>Level</th>
<th>Slots</th>
<th>Prev</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>#</td>
<td>��</td>
<td>��</td>
</tr>
</tbody>
</table>

Sorted Keys

<table>
<thead>
<tr>
<th>K1</th>
<th>K2</th>
<th>K3</th>
<th>K4</th>
<th>K5</th>
<th>...</th>
<th>Kn</th>
</tr>
</thead>
</table>

Values

| ��   | ��   | ��   | ��   | �� | ... | ��   |
B+TREE DESIGN CHOICES

Non-Unique Indexes: One key maps to multiple values.

Variable Length Keys: The size of each key is not the same.
B+TREE: NON-UNIQUE INDEXES

Approach #1: Duplicate Keys
→ Use the same leaf node layout but store duplicate keys multiple times.

Approach #2: Value Lists
→ Store each key only once and maintain a linked list of unique values.
B+TREE: DUPLICATE KEYS

B+Tree Leaf Node

Sorted Keys

Values
B+TREE: DUPLICATE KEYS

B+Tree Leaf Node

- Sorted Keys: K1, K1, K1, K2, K2, ..., Kn
- Level: #, #, #
- Slots: #, #, #
- Prev: X
- Next: X
- Values: • • • • • •
B+TREE: DUPLICATE KEYS

B+Tree Leaf Node

Sorted Keys

Values

K1  K1  K1  K2  K2  ...  Kn

Level  Slots  Prev  Next
B+TREE: VALUE LISTS

B+Tree Leaf Node

- Level
- Slots
- Prev
- Next

Sorted Keys
- K1
- K2
- K3
- K4
- K5
- ... Kn

Values
- ...
- ...
- ...
- ...
- ...
- ...

...
B+TREE: VALUE LISTS

B+Tree Leaf Node

- Level
- Slots
- Prev
- Next

Sorted Keys

K1  K2  K3  K4  K5  \cdots  Kn

Values

\vdots
B+TREE: VARIABLE LENGTH KEYS

Approach #1: Pointers
→ Store the keys as pointers to the tuple’s attribute.

Approach #2: Variable Length Nodes
→ The size of each node in the b+tree can vary.
→ Requires careful memory management.

Approach #3: Key Map
→ Embed an array of pointers that map to the key + value list within the node.
**B+Tree: Key Map**

**B+Tree Leaf Node**

- **Level**
- **Slots**
- **Prev**
- **Next**

**Key Map**

- K1
- K2

**Key+Values**

- K1
- K2
- V1
- V2
- V3
B+TREE: KEY MAP

B+Tree Leaf Node

<table>
<thead>
<tr>
<th>Level</th>
<th>Slots</th>
<th>Prev</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>#</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Key Map

<table>
<thead>
<tr>
<th>Key Map</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

Key+Values

<table>
<thead>
<tr>
<th>Key+Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>K1 V1 V2</td>
</tr>
<tr>
<td>K2 V1 V2 V3</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
B+TREE ALTERNATIVES

T-Trees
Skip Lists
Radix Trees (aka Patricia Trees)
MassTree
Fractal Trees
B+TREE ALTERNATIVES

- T-Trees
- Skip Lists
- Radix Trees (aka Patricia Trees)
- MassTree
- Fractal Trees
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

Diagram of T-trees showing the structure and relationships between the nodes.
T-TREES

T-Tree Node

Min-K

Max-K

Parent Pointer

Left Child Pointer

Right Child Pointer
T-TREES

T-Tree Node

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

**T-Tree Node**

- **Parent Pointer**
- **Node Boundaries**
- **Min-K**
- **Max-K**
- **Left Child Pointer**
- **Right Child Pointer**
T-TREES

Key Space (Low→High)

1 2 3 4 5 6 7

T-Tree Node

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-Tree Node**

- **Parent Pointer**
- **Left Child Pointer**
- **Right Child Pointer**
- **Min-K**
- **Max-K**
**T-TREES**

**Key Space (Low→High)**

1 2 3 4 5 6 7

T-TREE Node

- **Parent Pointer**
- **Min-K**
- **Max-K**
- **Left Child Pointer**
- **Right Child Pointer**
Advantages
→ Uses less memory because it does not store keys inside of each node.

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

- **Levels**
  - P=N
  - P=N/2
  - P=N/4
  - K1
  - V1
  - K2
  - V2
  - K3
  - V3
  - K4
  - V4
  - K6
  - V6

- **End**
  - ∞
SKIP LISTS: INSERT

Levels

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

End

- $\infty$

Nodes:
- K1, V1
- K2, V2
- K3, V3
- K4, V4
- K6, V6
**SKIP LISTS: INSERT**

**Levels**

- **P=N/4**
  - K1
  - V1

- **P=N/2**
  - K2
  - V2
  - K3
  - V3

- **P=N**
  - K4
  - V4

**End**

- ∞
  - K6
  - V6
SKIP LISTS: INSERT

Levels

P=N/4

P=N/2

P=N

End

∞

∞

∞

∞

∞

∞

∞
SKIP LISTS: INSERT

**Levels**

- **P=N**
  - K1: V1
  - K2: V2

- **P=N/2**
  - K2

- **P=N/4**
  - K1

**End**

- ∞

- K4

- K6: V6
**SKIP LISTS: INSERT**

**Txn #1: Insert K5**

```
Levels

K1 V1
P=N

K2 V2
P=N/2

K2
P=N/4

K3 V3

K4 V4

K4

K6 V6

End

∞

∞

∞
```
SKI P LISTS: INSERT

Txn #1: Insert K5
**Txn #1: Insert K5**

**Levels**

- **P=N**: K1 \(\rightarrow\) V1
- **P=N/2**: K2 \(\rightarrow\) V2
- **P=N/4**: K3 \(\rightarrow\) V3
- **∞**: K4 \(\rightarrow\) V4

**End**

- K5 \(\rightarrow\) V5
- K6 \(\rightarrow\) V6
- ∞
**SKIP LISTS: INSERT**

**Txn #1: Insert K5**

**Levels**

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

**End**

- $\infty$

**Diagram:**
- Node K1 with value V1
- Node K2 with value V2
- Node K3 with value V3
- Node K4 with value V4
- Node K5
- Node K6 with value V6

The transaction involves inserting key K5 into the skip list.
**SKIP LISTS: INSERT**

**Levels**

- **P=N/4**
  - $K_5$

- **P=N/2**
  - $K_2$
  - $V_2$

- **P=N**
  - $K_1$
  - $V_1$

**End**

- $\infty$

**Txn #1: Insert K5**
SKIP LISTS: SEARCH

Levels

P=N/4

K1
V1

K2
V2

K3
V3

K4
V4

K5
V5

K6
V6

P=N/2

P=N

End

∞

∞

∞
**SKIP LISTS: SEARCH**

**Txn #1: Find K3**
SKIP LISTS: SEARCH

Txn #1: Find K3
**SKIP LISTS: SEARCH**

**Txn #1: Find K3**

Levels

- **P=N**: K1, V1
- **P=N/2**: K2, V2
- **P=N/4**: K3, V3
- **K3<K5**: K4, V4
- **K3<K5**: K5, V5
- **K3<K5**: K6, V6

End

- **∞**: K5
- **∞**: K6
- **∞**: K7

Diagram showing the search process for K3 in a skip list structure.
**SKIP LISTS: SEARCH**

**Txn #1: Find K3**

Levels

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

**K3 < K5**

- K3 < K5
- K3 < K2

**K3 > K2**

- K3 > K2

End

- K5
- K6
- ∞
**Levels**

- **P=N/4**: K3<K5
- **P=N/2**: K3>K2, K3<K4
- **P=N**: K3<K5

**Txn #1: Find K3**

Steps:
1. **P=N**: K3<K5 (Skip List)
2. **P=N/2**: K3>K2, K3<K4 (Skip List)
3. **End**: K5, K6, Infinity
SKIP LISTS: SEARCH

Txn #1: Find K3

Levels

P=N

K1 V1

P=N/2

K2 V2

K3<K5

P=N/4

K2

K3>K2

K3<K4

End

K5

K5

K6

∞

∞

∞

∞
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.

Disadvantages
→ Not cache friendly because they do not optimize locality of references.
→ Reverse search is non-trivial.
SKIP LISTS: REVERSE SEARCH

Source: MemSQL
**Levels**

- **P=N**: 
  - K1
  - V1

- **P=N/2**: 
  - K2
  - V2

- **P=N/4**: 
  - K3
  - V3

**End**

- ∞

---

**TXN #1: Find K3**

Source: MemSQL
SKIP LISTS: REVERSE SEARCH

**Txn #1: Find K3**

Levels

<table>
<thead>
<tr>
<th>Levels</th>
<th>P=N</th>
<th>P=N/2</th>
<th>P=N/4</th>
</tr>
</thead>
<tbody>
<tr>
<td>K1 / V1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K2 / V2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K3 / V3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K4 / V4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K5</td>
<td>K5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>K6 / V6</td>
<td>K6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

End

Source: MemSQL

CMU 15-721 (Spring 2016)
**SKIP LISTS: REVERSE SEARCH**

**Levels**

1. $P = N$:
   - $K_1$ with $V_1$
   - $K_2$ with $V_2$

2. $P = N/2$:
   - $K_2$ with $V_2$
   - $K_3$ with $V_3$

3. $P = N/4$:
   - $K_3$ with $V_3$
   - $K_4$ with $V_4$

**Txn #1: Find $K_3$**

- Begin at $P = N$ level.
- Move down to $P = N/2$ level.
- Move down to $P = N/4$ level.
- Stop at $K_3$.

**End**

- Move to $P = N/4$ level.
- Move to $P = N/2$ level.
- Move to $P = N$ level.
- Stop at $K_5$.

Source: MemSQL

CMU 15-721 (Spring 2016)
**Levels**

- **P=N**
  - **K1** (V1)
  - **K2** (V2)
  - **K3** (V3)

- **P=N/2**
  - **K2**
  - **K3**

- **P=N/4**
  - **K3**

**End**

- **K5**
  - **K6**

**Txn #1: Find K3**

Source: MemSQL
**Levels**

- **$P=N$**
  - $K_1$, $V_1$
  - $K_2$, $V_2$
  - $K_3$, $V_3$
- **$P=N/4$**
  - $K_2$, $V_2$
  - $K_3$, $V_3$
- **$P=N/2$**
  - $K_2$, $V_2$
  - $K_4$, $V_4$
  - $K_5$, $V_5$
  - $K_6$, $V_6$

**End**

- **$\infty$**
- Last $K_2$
- Last $K_4$
- Last $K_5$

---

**SKIP LISTS: REVERSE SEARCH**

**Txn #1: Find K3**

Source: MemSQL
**Levels**

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

**Txn #1: Find K3**

- **K3 < K5**
- **K5**

**End**

- LAST K2
- LAST K4
- LAST K5
**SKIP LISTS: REVERSE SEARCH**

**Txn #1: Find K3**

- **Levels**
  - \( P = N \)
  - \( P = N/2 \)
  - \( P = N/4 \)

- **Source:** MemSQL

- **End**
  - \( \infty \)
  - **LAST:** K2, K4, K5

- **Comparison:**
  - K3 < K5
  - K3 < K4
SKIP LISTS: REVERSE SEARCH

**Txn #1: Find K3**

```
Levels

P=N/4

P=N/2

P=N

K1  V1

K2  V2

K2

K3  V3

K4  V4

K4

K5

End

∞

∞

∞

LAST K2

LAST K4

LAST K5

K3<K2

K3<K4

K3<K5

Source: MemSQL
```

**Notes:**
- The skip list is constructed with multiple levels.
- The search starts at the top level and moves down through the levels.
- The elements K1, K2, K3, K4, K5, and K6 are shown, with values V1, V2, V3, V4, V5, and V6.
- The search for K3 results in the following comparisons:
  - K3 < K5
  - K3 < K4
  - K3 > K2

**Diagram:**
- The diagram illustrates the skip list structure and the search process.
Levels

Txn #1: Find K3

Source: MemSQL
Skip Lists: Reverse Search

**Txn #1: Find K3**

Levels

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

Txn #1:

1. Find K3
2. K3 < K5
3. K3 < K4
4. K3 > K2

End

- LAST K2
- LAST K4
- LAST K5

Source: MemSQL

CMU 15-721 (Spring 2016)
WHY ARE INDEXES DIFFERENT?

The DBMS has to treat locking in indexes differently than how its concurrency control scheme manages database objects.

The physical structure can change as long as the logical contents are consistent.
PROBLEM SCENARIO #1
PROBLEM SCENARIO #1

Txn #1: Check if 25 exists
PROBLEM SCENARIO #1

Txn #1: Check if 25 exists
PROBLEM SCENARIO #1

Txn #1: Check if 25 exists
PROBLEM SCENARIO #1

Txn #1: Check if 25 exists
**Problem Scenario #1**

**Txn #1:** Check if 25 exists
PROBLEM SCENARIO #1

Txn #1: Check if 25 exists
PROBLEM SCENARIO #1

Txn #1: Check if 25 exists
**PROBLEM SCENARIO #1**

**Txn #1:** Check if 25 exists
PROBLEM SCENARIO #1

**Txn #1**: Check if 25 exists
**PROBLEM SCENARIO #1**

**Txn #1:** Check if 25 exists  
**Txn #2:** Insert 25
PROBLEM SCENARIO #1

Txn #1: Check if 25 exists
Txn #2: Insert 25
PROBLEM SCENARIO #1

Txn #1: Check if 25 exists
Txn #2: Insert 25
PROBLEM SCENARIO #1

Txn #1: Check if 25 exists
Txn #2: Insert 25
**PROBLEM SCENARIO #1**

- **Txn #1:** Check if 25 exists
- **Txn #2:** Insert 25
- **Txn #1:** Insert 25
PROBLEM SCENARIO #1

**Txn #1**: Check if 25 exists
**Txn #2**: Insert 25
**Txn #1**: Insert 25
PROBLEM SCENARIO #2
**PROBLEM SCENARIO #2**

**Txn #1:** Scan [12, 23]
PROBLEM SCENARIO #2

Txn #1: Scan [12, 23]
PROBLEM SCENARIO #2

Txn #1: Scan [12, 23]
PROBLEM SCENARIO #2

Txn #1: Scan [12, 23]
Txn #2: Insert 21
PROBLEM SCENARIO #2

Txn #1: Scan [12, 23]
Txn #2: Insert 21
PROBLEM SCENARIO #2

Txn #1: Scan [12, 23]
Txn #2: Insert 21
**PROBLEM SCENARIO #2**

**Txn #1:** Scan $[12, 23]$

**Txn #2:** Insert 21
**PROBLEM SCENARIO #2**

- **Txn #1**: Scan [12, 23]
- **Txn #2**: Insert 21
- **Txn #1**: Scan [12, 23]
PROBLEM SCENARIO #2

Txn #1: Scan [12, 23]
Txn #2: Insert 21
Txn #1: Scan [12, 23]
LOCKS VS. LATCHES

Locks
→ Protects the index’s logical contents from other txns.
→ Held for txn duration.
→ Need to be able to rollback changes.

Latches
→ Protects the critical sections of the index’s internal data structure from other threads.
→ Held for operation duration.
→ Do not need to be able to rollback changes.
## LOCKS VS. LATCHES

<table>
<thead>
<tr>
<th>locks</th>
<th>latches</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>separate</strong>...</td>
<td>Thread</td>
</tr>
<tr>
<td><strong>protect</strong>...</td>
<td>Database Contents</td>
</tr>
<tr>
<td><strong>during</strong>...</td>
<td>Critical Sections</td>
</tr>
<tr>
<td><strong>modes</strong>...</td>
<td>Read, Write</td>
</tr>
<tr>
<td><strong>deadlock</strong></td>
<td>Avoidance</td>
</tr>
<tr>
<td><strong>...by</strong>...</td>
<td>Coding Discipline</td>
</tr>
<tr>
<td><strong>kept in</strong>...</td>
<td>Protected Data Structure</td>
</tr>
</tbody>
</table>

*Source: Goetz Graefe*
INDEX LOCKS

Lock Table

txn1  txn2  txn3  ...
X      S      S

txn3  txn2  txn4  ...
S      S      S

txn4 txn6  txn5  ...
IX     X      S
INDEX LOCKS

Lock Table

txn1 X
txn2 S
txn3 S

• • •

txn2 S
txn3 S
txn4 S

• • •

txn3 S
txn4 S
txn5 S

• • •
INDEX LOCKS

Lock Table

- **txn1**: X
- **txn2**: S
- **txn3**: S
- **txn4**: S

- **txn3**: S
- **txn2**: S
- **txn4**: S

- **txn4**: IX
- **txn6**: X
- **txn5**: S
LOCK-FREE INDEXES

Possibility #1: No Locks
→ Txns don’t acquire locks to access/modify database.
→ Still have to use latches to install updates.

Possibility #2: No Latches
→ Use multi-versioning inside of the index. Swap pointers using atomic updates to install updates.
→ Still have to use locks to validate txns.
INDEX LOCKING

Predicate Locks
Key-Value Locks
Gap Locks
Key-Range Locks
Hierarchical Locking
PREDICATE LOCKS

Proposed locking scheme from System R.
→ Shared lock on the predicate in a WHERE clause of a SELECT query.
→ Exclusive lock on the predicate in a WHERE clause of any UPDATE, INSERT or DELETE query.

Never implemented in any system.
SELECT SUM(balance) 
FROM account 
WHERE name = ‘Tupac’

INSERT INTO account 
(name, balance) 
VALUES (‘Tupac’, 100);

Records in Table ‘account’
SELECT SUM(balance) FROM account WHERE name = 'Tupac'

INSERT INTO account (name, balance) VALUES ('Tupac', 100);

Records in Table 'account'
**PREDICATE LOCKS**

- **SELECT** SUM(balance) FROM account WHERE name = 'Tupac'

- **INSERT INTO** account (name, balance) **VALUES** ('Tupac', 100);

---

**Records in Table 'account'**

- name = 'Tupac'

---

CMU 15-721 (Spring 2016)
SELECT SUM(balance) FROM account WHERE name = 'Tupac'

INSERT INTO account (name, balance) VALUES ('Tupac', 100);

Records in Table 'account'

name='Tupac'

name='Tupac' ∧ balance=100
KEY-VALUE LOCKS

Locks that cover a single key value.
Need “virtual keys” for non-existent values.

B+Tree Leaf Node

10 12 14 16
KEY-VALUE LOCKS

Locks that cover a single key value.
Need “virtual keys” for non-existent values.

B+Tree Leaf Node

```
| 10 | 12 | 14 | 16 |
```
GAP LOCKS

Each txn acquires a key-value lock on the single key that it wants to access. Then get a gap lock on the next key gap.

B+Tree Leaf Node

10  12  14  16
GAP LOCKS

Each txn acquires a key-value lock on the single key that it wants to access. Then get a gap lock on the next key gap.

*B+Tree Leaf Node*

```
10 {Gap} 12 {Gap} 14 {Gap} 16
```
GAP LOCKS

Each txn acquires a key-value lock on the single key that it wants to access. Then get a gap lock on the next key gap.

B+Tree Leaf Node

<table>
<thead>
<tr>
<th>10</th>
<th>{Gap}</th>
<th>12</th>
<th>{Gap}</th>
<th>14</th>
<th>{Gap}</th>
<th>16</th>
</tr>
</thead>
</table>

Gap (14, 16)
A txn takes locks on ranges in the key space.
→ Each range is from one key that appears in the relation, to the next that appears.
→ Define lock modes so conflict table will capture commutativity of the operations available.
KEY-RANGE LOCKS

Locks that cover a key value and the gap to the next key value in a single index.
→ Need “virtual keys” for artificial values (infinity)

B+Tree Leaf Node

10 {Gap} 12 {Gap} 14 {Gap} 16
KEY-RANGE LOCKS

Locks that cover a key value and the gap to the next key value in a single index.
→ Need “virtual keys” for artificial values (infinity)

B+Tree Leaf Node

10  {Gap}  12  {Gap}  14  {Gap}  16

Next Key [14, 16]
KEY-RANGE LOCKS

Locks that cover a key value and the gap to the next key value in a single index.
→ Need “virtual keys” for artificial values (infinity)

B+Tree Leaf Node

10 \{Gap\} 12 \{Gap\} 14 \{Gap\} 16

Prior Key (12, 14)
HIERARCHICAL LOCKING

Allow for a txn to hold wider key-range locks with different locking modes.
→ Reduces the number of visits to lock manager.

B+Tree Leaf Node

10 {Gap} 12 {Gap} 14 {Gap} 16
HIERARCHICAL LOCKING

Allow for atxn to hold wider key-range locks with different locking modes.
→ Reduces the number of visits to lock manager.
Allow for a txn to hold wider key-range locks with different locking modes. → Reduces the number of visits to lock manager.
Hierarchical locking essentially provides predicate locking without complications.
→ Index locking occurs only in the leaf nodes.
→ Latching is to ensure consistent data structure.

Just like concurrency control schemes, research on fast indexes is hot again.
PRISON TATTOOS

Some of you are going to end up in prison. → This is just the nature of the database game.

Part of surviving prison is being able to navigate and avoid the various factions.
TEAR DROP
THREE DOTS
FIVE DOTS
MARA SALVATRUCHA GANG (MS13)
ARYAN BROTHERHOOD
NEXT CLASS

Bw-Tree (Hekaton)
Concurrent Skip Lists (MemSQL)
ART Index (HyPer)