08 Tree Indexes – Part II
UPCOMING DATABASE EVENTS

Vertica Talk
→ Monday Sep 23rd @ 4:30pm
→ GHC 8102
TODAY'S AGENDA

More B+Trees
Additional Index Magic
Tries / Radix Trees
Inverted Indexes
B+TREE: DUPLICATE KEYS

Approach #1: Append Record Id
→ Add the tuple's unique record id as part of the key to ensure that all keys are unique.
→ The DBMS can still use partial keys to find tuples.

Approach #2: Overflow Leaf Nodes
→ Allow leaf nodes to spill into overflow nodes that contain the duplicate keys.
→ This is more complex to maintain and modify.
B+TREE: APPEND RECORD ID

<Key,RecordId>
**B+TREE: APPEND RECORD ID**

Insert 6

```
<Key,RecordId>
```

Diagram: A B+Tree structure showing the insertion of the key 6.
B+TREE: APPEND RECORD ID

Insert <6,(Page,Slot)>
Insert \( <6, (\text{Page, Slot})> \)
**B+TREE: APPEND RECORD ID**

Insert `<6, (Page, Slot)>`

```
<5
```

```
<9
```

```
<Key, RecordId>
```

```
1 3
```

```
5 9
```

```
7 8
```

```
9 13
```
B+TREE: APPEND RECORD ID

Insert <6,(Page,Slot)>

<Key,RecordId>
B+TREE: APPEND RECORD ID

Insert <6, (Page, Slot)>

<Key, RecordId>
Insert 6
B+TREE: OVERFLOW LEAF NODES

Insert 6

Insert 7
B+TREE: OVERFLOW LEAF NODES

Insert 6
Insert 7
Insert 6
DEMO

B+Tree vs. Hash Indexes
Table Clustering
Most DBMSs automatically create an index to enforce integrity constraints but not referential constraints (foreign keys).
→ Primary Keys
→ Unique Constraints

```
CREATE TABLE foo (  
id SERIAL PRIMARY KEY,
val1 INT NOT NULL,
val2 VARCHAR(32) UNIQUE  
);

CREATE UNIQUE INDEX foo_pkey ON foo (id);

CREATE UNIQUE INDEX foo_val2_key ON foo (val2);
```
IMPLICIT INDEXES

Most DBMSs automatically create an index to enforce integrity constraints but not referential constraints (foreign keys).

→ Primary Keys
→ Unique Constraints

```
CREATE TABLE foo (  
id SERIAL PRIMARY KEY,  
val1 INT NOT NULL,  
val2 VARCHAR(32) UNIQUE  
);
```

```
CREATE TABLE bar (  
id INT REFERENCES foo (val1),  
val VARCHAR(32)  
);
```
Most DBMSs automatically create an index to enforce integrity constraints but not referential constraints (foreign keys).

→ Primary Keys
→ Unique Constraints

```sql
CREATE TABLE foo (  
id SERIAL PRIMARY KEY,  
val1 INT NOT NULL,  
val2 VARCHAR(32) UNIQUE  
);

CREATE TABLE bar (  
id INT REFERENCES foo (val1),  
val VARCHAR(32)  
);

CREATE INDEX foo_val1_key  
ON foo (val1);
```
Most DBMSs automatically create an index to enforce integrity constraints but not referential constraints (foreign keys).
→ Primary Keys
→ Unique Constraints

```sql
CREATE TABLE foo (
  id SERIAL PRIMARY KEY,
  val1 INT NOT NULL,
  val2 VARCHAR(32) UNIQUE
);

CREATE INDEX foo_val1_key
  ON foo (val1);

CREATE TABLE bar (
  id INT REFERENCES foo (val1),
  val VARCHAR(32)
);
```
IMPLICIT INDEXES

Most DBMSs automatically create an index to enforce integrity constraints but not referential constraints (foreign keys).

→ Primary Keys
→ Unique Constraints

```
CREATE TABLE foo (  
id SERIAL PRIMARY KEY,  
val1 INT NOT NULL UNIQUE,  
val2 VARCHAR(32) UNIQUE
);
```

```
CREATE TABLE bar (  
id INT REFERENCES foo (val1),  
val VARCHAR(32)
);
```
PARTIAL INDEXES

Create an index on a subset of the entire table. This potentially reduces its size and the amount of overhead to maintain it.

One common use case is to partition indexes by date ranges.
→ Create a separate index per month, year.

CREATE INDEX idx_foo
    ON foo (a, b)
    WHERE c = 'WuTang';
PARTIAL INDEXES

Create an index on a subset of the entire table. This potentially reduces its size and the amount of overhead to maintain it.

One common use case is to partition indexes by date ranges.
→ Create a separate index per month, year.

```
CREATE INDEX idx_foo
    ON foo (a, b)
    WHERE c = 'WuTang';

SELECT b FROM foo
WHERE a = 123
    AND c = 'WuTang';
```
If all the fields needed to process the query are available in an index, then the DBMS does not need to retrieve the tuple.

This reduces contention on the DBMS's buffer pool resources.

CREATE INDEX idx_foo
ON foo (a, b);

SELECT b FROM foo
WHERE a = 123;
If all the fields needed to process the query are available in an index, then the DBMS does not need to retrieve the tuple.

This reduces contention on the DBMS's buffer pool resources.
INDEX INCLUDE COLUMNS

Embed additional columns in indexes to support index-only queries.

These extra columns are only stored in the leaf nodes and are not part of the search key.

```sql
CREATE INDEX idx_foo
  ON foo (a, b)
  INCLUDE (c);
```
INDEX INCLUDE COLUMNS

Embed additional columns in indexes to support index-only queries.

These extra columns are only stored in the leaf nodes and are not part of the search key.

```
CREATE INDEX idx_foo
    ON foo (a, b)
    INCLUDE (c);
```

```
SELECT b FROM foo
    WHERE a = 123
    AND c = 'WuTang';
```
INDEX INCLUDE COLUMNS

Embed additional columns in indexes to support index-only queries.

These extra columns are only stored in the leaf nodes and are not part of the search key.

```sql
CREATE INDEX idx_foo ON foo (a, b) INCLUDE (c);

SELECT b FROM foo WHERE a = 123 AND c = 'WuTang';
```
INDEX INCLUDE COLUMNS

Embed additional columns in indexes to support index-only queries.

These extra columns are only stored in the leaf nodes and are not part of the search key.

```
CREATE INDEX idx_foo ON foo (a, b)
   INCLUDE (c);
```

```
SELECT b FROM foo
WHERE a = 123
   AND c = 'WuTang';
```
FUNCTIONAL/EXPRESSION INDEXES

An index does not need to store keys in the same way that they appear in their base table.

```
SELECT * FROM users
WHERE EXTRACT(dow FROM login) = 2;
```

```
CREATE INDEX idx_user_login
ON users (login);
```
FUNCTIONAL/EXPRESSION INDEXES

An index does not need to store keys in the same way that they appear in their base table.

```
SELECT * FROM users
WHERE EXTRACT(dow FROM login) = 2;
```

```
CREATE INDEX X_user_login ON users (login);
```

An index does not need to store keys in the same way that they appear in their base table.

You can use expressions when declaring an index.

```sql
CREATE INDEX idx_user_login ON users (EXTRACT(dow FROM login));
CREATE INDEX idx_user_login ON users (EXTRACT(dow FROM login));
SELECT * FROM users
WHERE EXTRACT(dow FROM login) = 2;
```
FUNCTIONAL/EXPRESSION INDEXES

An index does not need to store keys in the same way that they appear in their base table.

You can use expressions when declaring an index.

```sql
SELECT * FROM users
WHERE EXTRACT(dow FROM login) = 2;

CREATE INDEX idx_user_login
ON users (EXTRACT(dow FROM login));

CREATE INDEX idx_user_login
ON users (EXTRACT(dow FROM login));
```
FUNCTIONAL/EXPRESSION INDEXES

An index does not need to store keys in the same way that they appear in their base table.

You can use expressions when declaring an index.

```sql
SELECT * FROM users
WHERE EXTRACT(dow FROM login) = 2;

CREATE INDEX idx_user_login
ON users (EXTRACT(dow FROM login));

CREATE INDEX idx_user_login
ON users (EXTRACT(dow FROM login));

CREATE INDEX idx_user_login
ON users (EXTRACT(dow FROM login));

CREATE INDEX idx_user_login
ON foo (login)
WHERE EXTRACT(dow FROM login) = 2;
```
The inner node keys in a B+Tree cannot tell you whether a key exists in the index. You must always traverse to the leaf node.

This means that you could have (at least) one buffer pool page miss per level in the tree just to find out a key does not exist.
TRIE INDEX

Keys: HELLO, HAT, HAVE

Use a digital representation of keys to examine prefixes one-by-one instead of comparing entire key.

→ Also known as *Digital Search Tree*, *Prefix Tree*.
TRIE INDEX

Keys: HELLO, HAT, HAVE

Use a digital representation of keys to examine prefixes one-by-one instead of comparing entire key.

→ Also known as Digital Search Tree, Prefix Tree.
TRIE INDEX PROPERTIES

Shape only depends on key space and lengths.
→ Does not depend on existing keys or insertion order.
→ Does not require rebalancing operations.

All operations have $O(k)$ complexity where $k$ is the length of the key.
→ The path to a leaf node represents the key of the leaf
→ Keys are stored implicitly and can be reconstructed from paths.
The **span** of a trie level is the number of bits that each partial key / digit represents.

→ If the digit exists in the corpus, then store a pointer to the next level in the trie branch. Otherwise, store null.

This determines the **fan-out** of each node and the physical **height** of the tree.

→ *n*-way Trie = Fan-Out of *n*
TRIE KEY SPAN

1-bit Span Trie

- **K10**: 00000000 00010100
- **K25**: 00000000 00011001
- **K31**: 00000000 00011111
TRIE KEY SPAN

1-bit Span Trie

K10→ 00000000 00001010
K25→ 00000000 00011001
K31→ 00000000 00011111

Tuple Pointer
Node Pointer
Repeat 10x
TRIE KEY SPAN

1-bit Span Trie

Tuple Pointer  →  Node Pointer

Repeat 10x

K10 → 00000000  00001010
K25 → 00000000  00011001
K31 → 00000000  00011111
TRIE KEY SPAN

1-bit Span Trie

Tuple Pointer

Node Pointer

K10 $\rightarrow$ 00000000 00001010
K25 $\rightarrow$ 00000000 00011001
K31 $\rightarrow$ 00000000 00011111
TRIE KEY SPAN

1-bit Span Trie

K10 → 00000000 00001010
K25 → 00000000 00011001
K31 → 00000000 00011111
TRIE KEY SPAN

1-bit Span Trie

K10 → 00000000 00001010
K25 → 00000000 00011001
K31 → 00000000 00011111
TRIE KEY SPAN

1-bit Span Trie

K10→ 00000000 00001010
K25→ 00000000 00011001
K31→ 00000000 00011111
TRIE KEY SPAN

1-bit Span Trie

K10 → 00000000 00001010
K25 → 00000000 00011001
K31 → 00000000 00011111
TRIE KEY SPAN

1-bit Span Trie

Keys:
K10, K25, K31

K10 \rightarrow 00000000 00001010
K25 \rightarrow 00000000 00011001
K31 \rightarrow 00000000 00011111
1-bit Span Radix Tree

Omit all nodes with only a single child.
→ Also known as Patricia Tree.

Can produce false positives, so the DBMS always checks the original tuple to see whether a key matches.
RADIX TREE: MODIFICATIONS

Insert HAIR
RADIX TREE: MODIFICATIONS

Insert HAIR
RADIX TREE: MODIFICATIONS

- Insert HAIR
- Delete HAT
RADIX TREE: MODIFICATIONS

Insert HAIR
Delete HAT
RADIX TREE: MODIFICATIONS

- Insert HAIR
- Delete HAT
- Delete HAVE
RADIOX TREE: MODIFICATIONS

Insert HAIR
Delete HAT
Delete HAVE
RADIX TREE: MODIFICATIONS

Insert HAIR
Delete HAT
Delete HAVE
RADIX TREE: MODIFICATIONS

- Insert HAIR
- Delete HAT
- Delete HAVE
Not all attribute types can be decomposed into binary comparable digits for a radix tree.

→ **Unsigned Integers**: Byte order must be flipped for little endian machines.

→ **Signed Integers**: Flip two’s-complement so that negative numbers are smaller than positive.

→ **Floats**: Classify into group (neg vs. pos, normalized vs. denormalized), then store as unsigned integer.

→ **Compound**: Transform each attribute separately.
**RADIX TREE: BINARY COMPARABLE KEYS**

8-bit Span Radix Tree

```
0A

0B 0F

0B 0C 1D

0B 0D

0B 0D
```

Int Key: 168496141

Hex Key: 0A 0B 0C 0D

Little Endian

Big Endian

0D

0C

0B

0A

0D

0C

0B

0A
RADIX TREE: BINARY COMPARABLE KEYS

8-bit Span Radix Tree

Int Key: 168496141
Hex Key: 0A 0B 0C 0D

Find 658205
Hex 0A 0B 1D
RADIX TREE: BINARY COMPARABLE KEYS

8-bit Span Radix Tree

Int Key: 168496141
Hex Key: 0A 0B 0C 0D

Find 658205
Hex 0A 0B 1D

Little Endian

Big Endian
OBSERVATION

The tree indexes that we've discussed so far are useful for "point" and "range" queries:
→ Find all customers in the 15217 zip code.
→ Find all orders between June 2018 and September 2018.

They are not good at keyword searches:
→ Find all Wikipedia articles that contain the word "Pavlo"
CREATE TABLE useracct (  
  userID INT PRIMARY KEY,  
  userName VARCHAR UNIQUE,  
);  

CREATE TABLE pages (  
  pageID INT PRIMARY KEY,  
  title VARCHAR UNIQUE,  
  latest INT  
);  

CREATE TABLE revisions (  
  revID INT PRIMARY KEY,  
  userID INT REFERENCES useracct (userID),  
  pageID INT REFERENCES pages (pageID),  
  content TEXT,  
  updated DATETIME  
);
If we create an index on the content attribute, what does that do?

This doesn't help our query. Our SQL is also not correct...

```
CREATE INDEX idx_rev_cntnt
ON revisions (content);
```

```
SELECT pageID FROM revisions
WHERE content LIKE '%Pavlo%';
```
An **inverted index** stores a mapping of words to records that contain those words in the target attribute.

→ Sometimes called a **full-text search index**.
→ Also called a **concordance** in old (like really old) times.

The major DBMSs support these natively. There are also specialized DBMSs.
QUERY TYPES

Phrase Searches
→ Find records that contain a list of words in the given order.

Proximity Searches
→ Find records where two words occur within $n$ words of each other.

Wildcard Searches
→ Find records that contain words that match some pattern (e.g., regular expression).
DESIGN DECISIONS

Decision #1: What To Store
→ The index needs to store at least the words contained in each record (separated by punctuation characters).
→ Can also store frequency, position, and other meta-data.

Decision #2: When To Update
→ Maintain auxiliary data structures to "stage" updates and then update the index in batches.
CONCLUSION

B+Trees are still the way to go for tree indexes.

Inverted indexes are covered in CMU 11-442.

We did not discuss geo-spatial tree indexes:
→ Examples: R-Tree, Quad-Tree, KD-Tree
→ This is covered in CMU 15-826.
NEXT CLASS

How to make indexes thread-safe!