Trees Indexes (Part II)
ADMINISTRIVIA

**Project #1** is due Wednesday Sept 26th @ 11:59pm

**Homework #2** is due Friday Sept 28th @ 11:59pm

**Project #2** will be released on Wednesday Sept 26th. First checkpoint is due Monday Oct 8th.
TODAY'S AGENDA

Additional Index Usage
Skip Lists
Radix Trees
Inverted Indexes
IMPLICIT INDEXES

Most DBMSs automatically create an index to enforce integrity constraints.
→ Primary Keys
→ Unique Constraints
→ Foreign Keys (?)

```
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);
```
Most DBMSs automatically create an index to enforce integrity constraints.

→ Primary Keys
→ Unique Constraints
→ Foreign Keys (?)

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

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

```sql
CREATE INDEX foo_val1_key
ON foo (val1);
```
IMPLICIT INDEXES

Most DBMSs automatically create an index to enforce integrity constraints.

→ Primary Keys
→ Unique Constraints
→ Foreign Keys (?)

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));
IMPLICIT INDEXES

Most DBMSs automatically create an index to enforce integrity constraints.
→ Primary Keys
→ Unique Constraints
→ Foreign Keys (?)

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 of 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. Not part of the search key.

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

Embed additional columns in indexes to support index-only queries. 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. 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';
```
The 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;
```
FUNCTIONAL/EXPRESSION INDEXES

The 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

The 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;
```
FUNCTIONAL/EXPRESSION INDEXES

The 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.

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

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

```
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

The 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

The 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.

```
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 foo (login)
WHERE EXTRACT(dow FROM login) = 2;
```
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 **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: $O(N)$
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
SKIP LISTS: EXAMPLE

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

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

P=N

P=N/2

P=N/4

K1
V1

K2
V2

K3
V3

K4
V4

K6
V6
SKIP LISTS: EXAMPLE

Levels

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

K1 \( \rightarrow \) K2 \( \rightarrow \) K3 \( \rightarrow \) K4 \( \rightarrow \) K6

V1 \( \rightarrow \) V2 \( \rightarrow \) V3 \( \rightarrow \) V4 \( \rightarrow \) V6

End

- \( \infty \)
- \( \infty \)
- \( \infty \)
SKIP LISTS: EXAMPLE

Levels

P=N

\[\begin{align*}
K1 & \rightarrow K2 & \rightarrow K3 & \rightarrow K4 & \rightarrow K6 \\
V1 & \rightarrow V2 & \rightarrow V3 & \rightarrow V4 & \rightarrow V6 \\
\end{align*}\]

P=N

P=N/2

P=N/4

End

\[\infty\]
SKIP LISTS: EXAMPLE

Levels

P=N/4

P=N/2

P=N

End

K1

V1

K2

V2

K3

V3

K4

V4

K6

V6

∞

∞

∞

P=N

P=N/2

P=N/4
SKIP LISTS: EXAMPLE

Levels

- $P=N$ (V1)
- $P=N/2$ (K1, V2)
- $P=N/4$ (K2, V3)

End

- $\infty$ (K4, V4)
- $\infty$ (K6, V6)
Insert K5
Skip Lists: Insert

Insert K5
SKIP LISTS: INSERT

Insert K5

Levels

- \( P = \frac{N}{4} \)
- \( P = \frac{N}{2} \)
- \( P = N \)

End

- \( K5 \)
- \( K6 \)
- \( \infty \)

- \( K1 \) → \( K2 \) → \( K3 \) → \( K4 \) → \( K5 \) → \( K6 \)
Insert K5

Skip Lists: Insert

Levels

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

Insert K5
**SKIP LISTS: INSERT**

Insert K5

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

- End
  - $\infty$

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

- Skip list pointers:
  - K5 connects to K6
  - K4 connects to K5
  - K3 connects to K4
  - K2 connects to K3
  - K1 connects to K2
  - $\infty$ connects to K5
**SKIP LISTS: INSERT**

Insert K5

---

**Levels**

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

**End**

- **∞**
- **∞**
- **∞**
SKIP LISTS: SEARCH

Find K3
SKIP LISTS: SEARCH

Find K3

K3 < K5

Levels

End

P = N

P = N/2

P = N/4

K1 V1

K2 V2

K3 V3

K4 V4

K5 V5

K6 V6

∞

∞

∞

∞
**SKIP LISTS: SEARCH**

Find K3

Levels

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

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

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

End

- **K5**
- **V5**

- **K6**
- **V6**

References:

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

Find K3

K3<K5

K3>K2

K3<K4

Levels

End

P=N/4

P=N/2

P=N

K3

K2

K1

V1

K2

V2

K3

V3

K4

V4

K5

V5

K6

V6

∞

∞

∞
SKIP LISTS: SEARCH

Find K3

Levels

P=N

K1
V1

K2
V2

K2

K3
V3

K3
K5

K3<K4

K3<K5

K3<K4

K3>K2

K3>K2

K5

K5

K5

K5

K5

K6
V6

End

P=N/4

P=N/2

P=N

∞

∞

∞
SKIP LISTS: DELETE

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.
SKIP LISTS: DELETE

Delete K5
SKIP LISTS: DELETE

Delete K5
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

K5

K5

K6 V6 Del false

End

∞
**SKIP LISTS: DELETE**

Delete K5

---

Levels

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

End

- **∞**: K5
- **∞**: K6
- **∞**: K1, V1
- **∞**: K2, V2
- **∞**: K3, V3
- **∞**: K4, V4
SKIP LISTS: DELETE

Delete K5

Levels

P=N/4

P=N/2

P=N

End

K1 Del false
V1

K2 Del false
V2

K3 Del false
V3

K4 Del false
V4

K5 Del true
V5

K6 Del false
V6

P=N

P=N/2

P=N/4
SKIP LISTS: DELETE

Delete K5

Levels

P=N/4

P=N/2

P=N

K1 V1 Del false
K2 V2 Del false
K3 V3 Del false
K4 V4 Del false
K5 Del false
K6 V6 Del false

End

P=N

P=N/2

P=N/4

∞

∞

∞

∞
SKIP LISTS

Advantages:
→ Uses less memory than a typical B+Tree if you don’t include reverse pointers.
→ Insertions and deletions do not require rebalancing.

Disadvantages:
→ Not disk/cache friendly because they do not optimize locality of references.
→ Reverse search is non-trivial.
RADIX TREE

Represent keys as individual digits. This allows threads to examine prefixes one-by-one instead of comparing entire key.
→ The height of the tree depends on the length of keys.
→ Does not require rebalancing
→ The path to a leaf node represents the key of the leaf
→ Keys are stored implicitly and can be reconstructed from paths.
TRIE VS. RADIX TREE

Trie

Keys: HELLO, HAT, HAVE
TRIE VS. RADIX TREE

Trie

Keys: HELLO, HAT, HAVE
TRIE VS. RADIX TREE

Trie

Keys: HELLO, HAT, HAVE
TRIE VS. RADIX TREE

Trie

Radix Tree

Keys: HELLO, HAT, HAVE
TRIE VS. RADIX TREE

**Trie**

- Keys: HELLO, HAT, HAVE

**Radix Tree**

- Keys: HELLO, HAT, HAVE
RADIX TREE: MODIFICATIONS
RADIX TREE: MODIFICATIONS

Insert HAIR
RADIX TREE: MODIFICATIONS

Insert HAIR
RADIX TREE: MODIFICATIONS

Insert HAIR
Delete HAT, HAVE
**RADIX TREE: MODIFICATIONS**

- **Insert**: HAIR
- **Delete**: HAT, HAVE
RADIX TREE: MODIFICATIONS

- **Insert** HAIR
- **Delete** HAT, HAVE
RADIX TREE: MODIFICATIONS

Insert HAIR
Delete HAT, 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**

**Int Key:** 168496141

**Hex Key:** 0A 0B 0C 0D

**Big Endian**

0A 0B 0C 0D

**Little Endian**

0A 0B 0C 0D
RADIX TREE: BINARY COMPARABLE KEYS

Int Key: 168496141

Hex Key: 0A 0B 0C 0D

Big Endian

Little Endian
RADIX TREE: BINARY COMPARABLE KEYS

Int Key: 168496141

Hex Key: 0A 0B 0C 0D

Big Endian

Little Endian
IN-MEMORY TABLE INDEXES

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

Source: Ziqi Wang
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
        REFERENCES revisions (revID),
);

CREATE TABLE revisions (  
    revID INT PRIMARY KEY,  
    userID INT REFERENCES useracct (userID),  
    pageID INT REFERENCES pages (pageID),  
    content TEXT,  
    updated DATETIME
);
WIKIPEDIA EXAMPLE

If we create an index on the content attribute, what does that actually 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!