Lecture 15: Trees (Part 2)

CREATING THE NEXT®

Administrivia

- Project presentations in class next week
- Assignment 3 due on Oct 24

Today's Agenda

Recap

B+Tree

- A **B+Tree** is a self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in **O(log n)**.
  - Generalization of a **binary search tree** in that a node can have more than two children.
  - Optimized for disk storage (*i.e.*, read and write at page-granularity).

B+Tree Properties

- A B+Tree is an **M-way** search tree with the following properties:
  - It is perfectly balanced (*i.e.*, every leaf node is at the same depth).
  - Every node other than the root, is **at least half-full**: M/2-1 <= keys <= M-1
  - Every inner node with k keys has k+1 non-null children (**node pointers**)

Today's Agenda

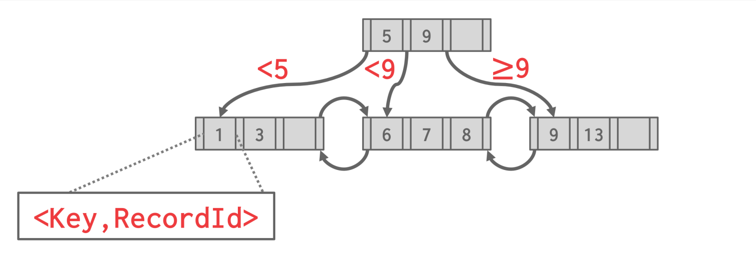
- More B+Trees
- Additional Index Magic
- Tries / Radix Trees
- Inverted Indexes

More B Trees

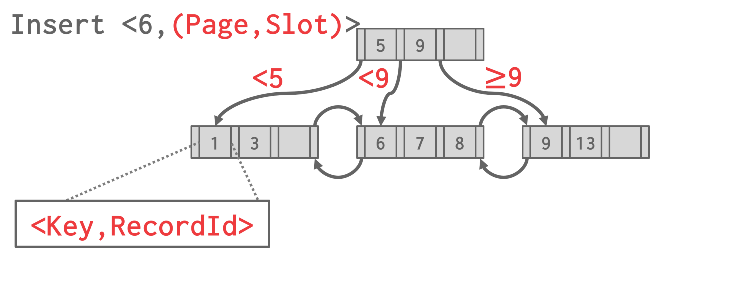
Duplicate Keys

- **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.

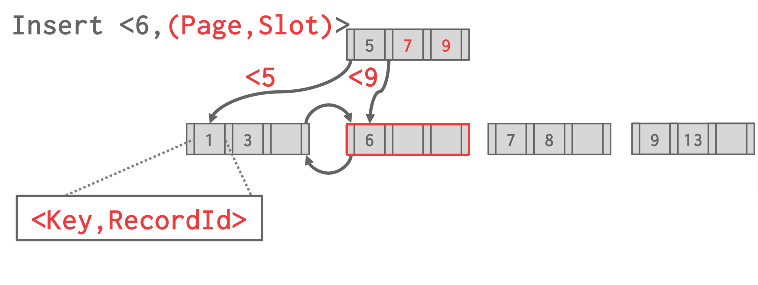
Append Record Id



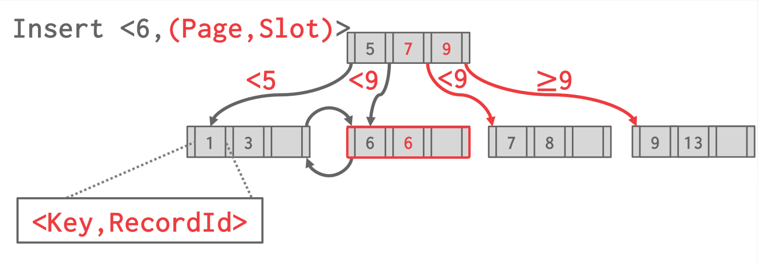
Append Record Id



Append Record Id



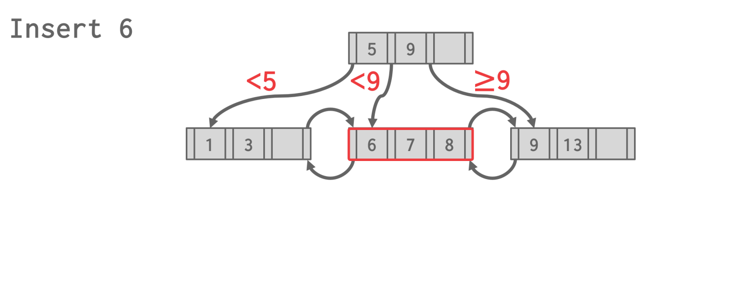
Append Record Id



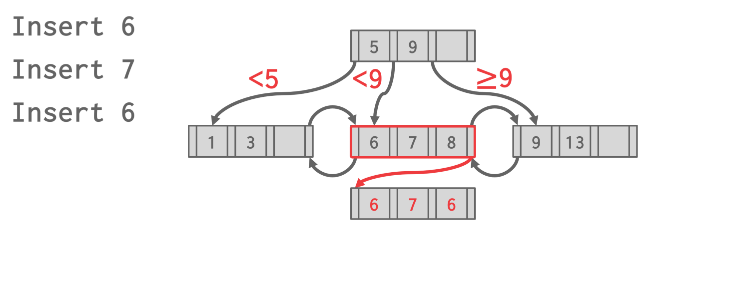
Duplicate Keys

- **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.

Overflow Leaf Nodes



Overflow Leaf Nodes



Partitioned B-Tree

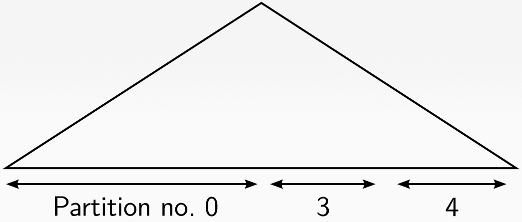
Bulk operations are fine if they are rare, but they are disruptive

- usually the B-tree has to be take offline
- the new cannot be queries easily
- existing queries must be halted

Partitioned B-Tree

Basic idea: *partition* the B-tree

- add an artificial column in front
- creates separate partitions with the B-tree



Partitioned B-Tree

Benefits:

- partitions are largely independent of each other
- one can append to the "rightmost" partition without disrupting the rest
- the index stays always online
- partitions can be merged lazily
- merge only when beneficial

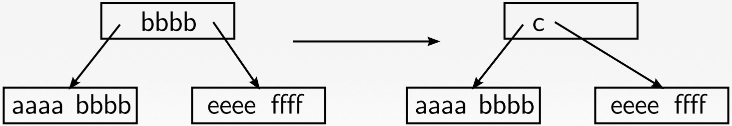
Drawbacks:

- no "global" order any more
- lookups have to access all partitions

Prefix B$^+$-tree

A B$^+$-tree can contain separators that do not occur in the data

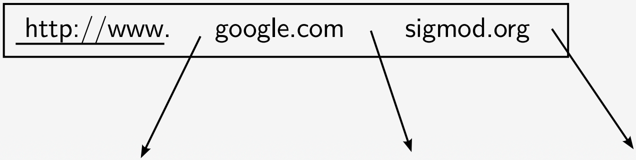
We can use this to save space:



- choose the smallest possible separator
- no change to the lookup logic is required

Prefix B$^+$-tree

We can do even better by factoring out a common prefix:



- only one prefix per page
- the change to the lookup logic is minor
- the lookup key itself is adjusted
- sometimes only inner nodes, to keep scans cheap

Prefix B^+-tree

The lexicographic sort order makes prefix compression attractive:

- neighboring entries tend to differ only at the end
- a common prefix occurs very frequently
- not only for strings, also for compound keys etc.
- in particular important if partitioned B-trees
- with big-endian ordering any value might get compressed

Additional Index Magic

Implicit Indexes

- Most DBMSs automatically create an index to enforce **integrity constraints**.
  - 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

- But, this is **not** done for **referential** integrity constraints (*i.e.*, foreign keys).

```
CREATE TABLE bar (
  id INT REFERENCES foo (val1),
  val VARCHAR(32)
);
CREATE INDEX foo_val1_key ON foo (val1); -- Not automatically done
```

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.

```sql
CREATE INDEX idx_foo ON foo (a, b)
  WHERE c = 'October';
SELECT b FROM foo WHERE a = 123 AND c = 'October';
```

Covering Indexes

- If all the fields needed to process the query are available in an index, then the DBMS does **not** need to retrieve the tuple from the heap.
- 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;
```

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 = 'October';
```

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 functions/expressions when declaring an index.

```
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.
- You can use functions/expressions when declaring an index.

```
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;
```

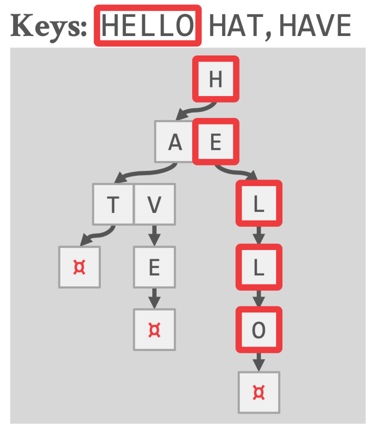
Tries / Radix Trees

Observation

- 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

- Use a **digital representation** of keys to examine prefixes one-by-one instead of comparing entire key.
  - *a.k.a.*, Digital Search Tree, Prefix Tree.



Properties

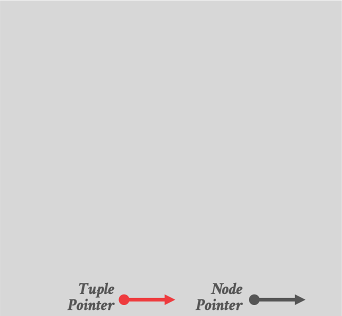
- Shape only depends on key space and lengths.
  - Does not depend on existing keys or insertion order.
  - Does not require rebalancing operations.
- All operations have **O(k)** complexity where **k** is the length of the key.
  - The path to a leaf node represents the key of the leaf
  - Keys are stored implicitly and can be reconstructed from paths.

Key Span

- 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.

Key Span

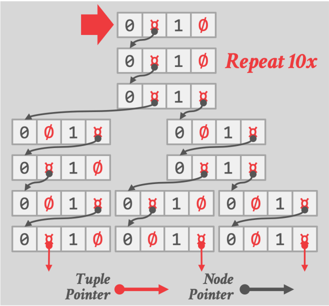
1-bit Span Trie




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

Key Span

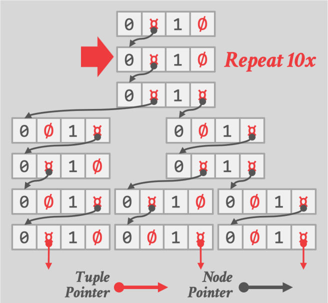
1-bit Span Trie




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

Key Span

1-bit Span Trie



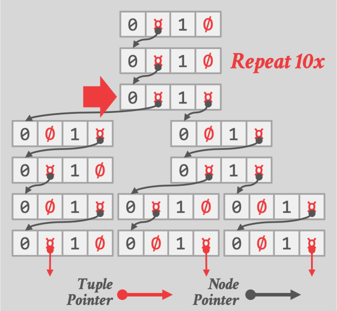
K10→ 00000000 00001010

K25→ 00000000 00011001

K31→ 00000000 00011111

Key Span

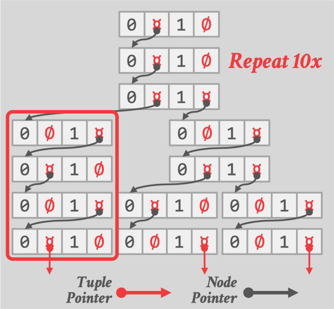
1-bit Span Trie




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

Key Span

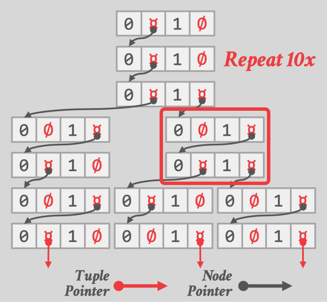
1-bit Span Trie




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

Key Span

1-bit Span Trie



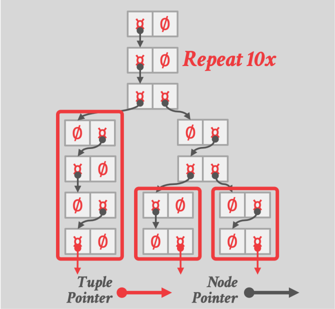
K10→ 00000000 00001010

K25→ 00000000 00011001

K31→ 00000000 00011111

Key Span

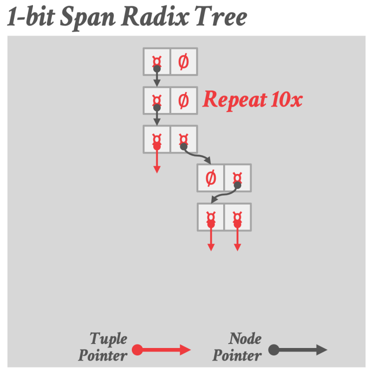
1-bit Span Trie



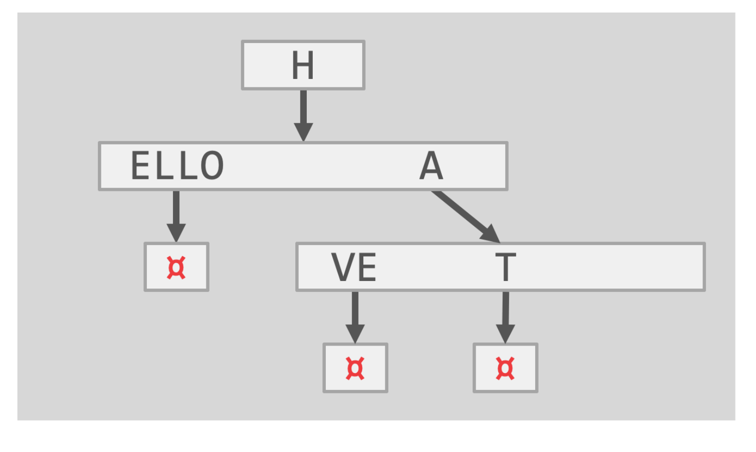

```
K10→ 00000000 00001010
K25→ 00000000 00011001
K31→ 00000000 00011111
```

Radix Tree

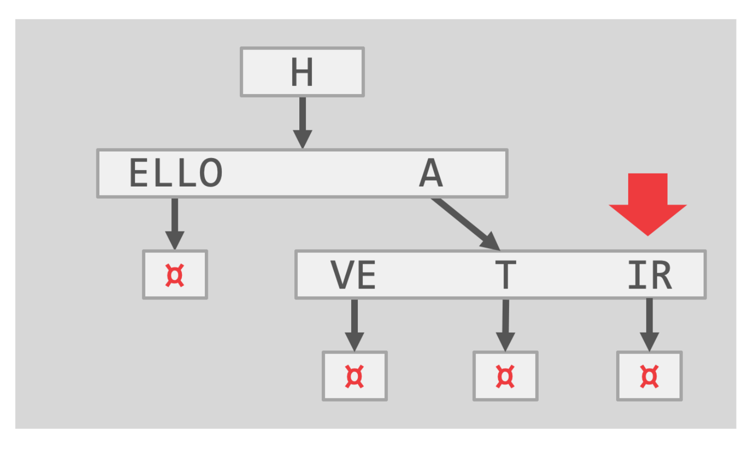
- Omit all nodes with **only** a single child.
  - *a.k.a.*, **Patricia Tree**.
- Can produce false positives
- So the DBMS always checks the original tuple to see whether a key matches.



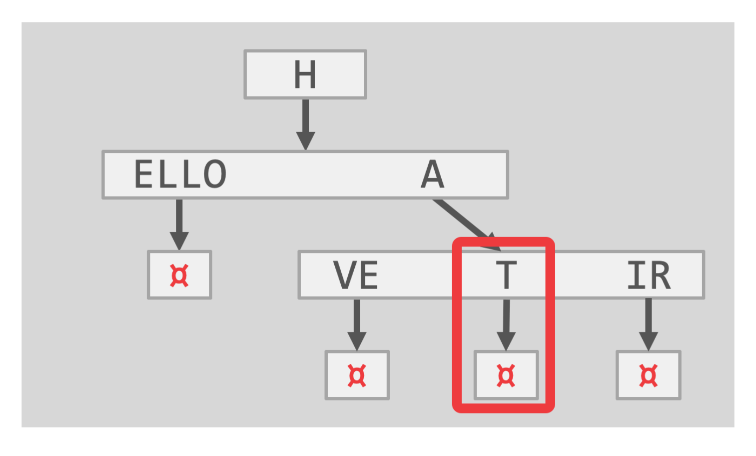
Radix Tree: Modifications



Radix Tree: Modifications

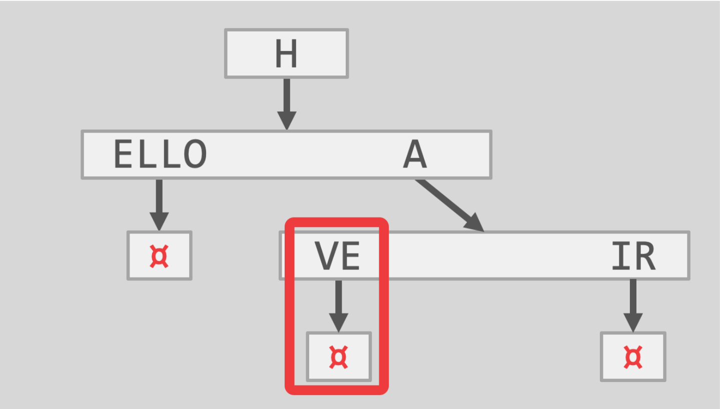


Radix Tree: Modifications

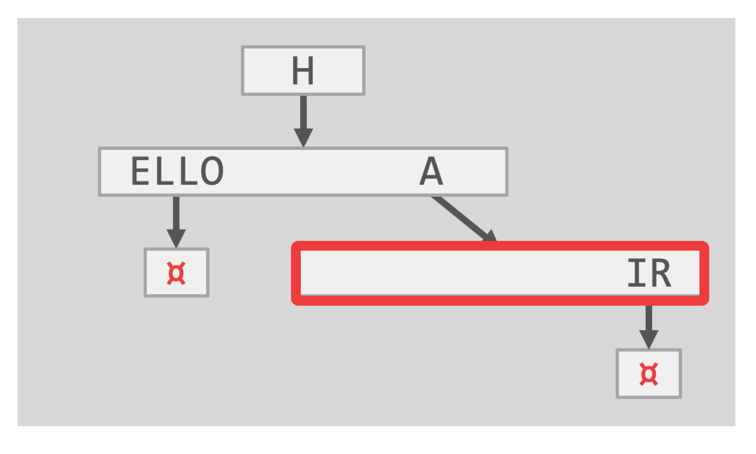


Georgia Tech

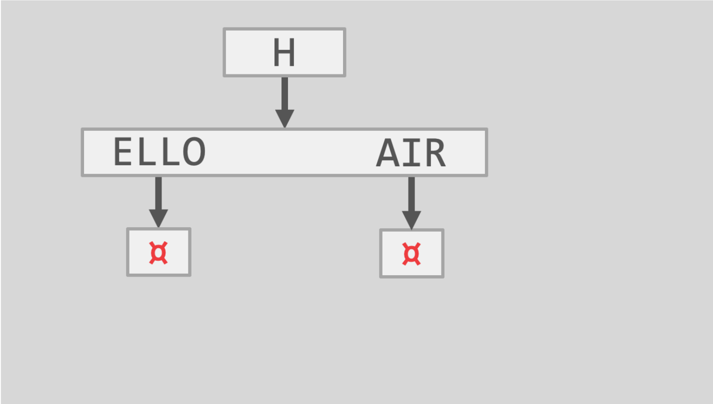
Radix Tree: Modifications



Radix Tree: Modifications



Radix Tree: Modifications

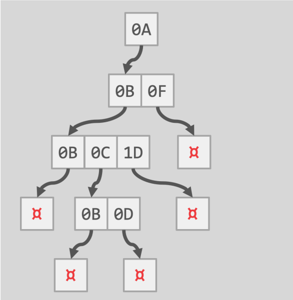


Radix Tree: Binary Comparable Keys

- 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



Int Key: `168496141`

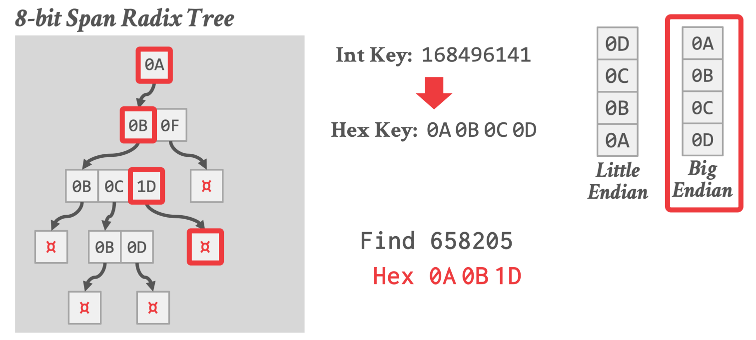
Hex Key: `0A 0B 0C 0D`

Little Endian	Big Endian
0D	0A
0C	0B
0B	0C
0A	0D

`Find 658205`

`Hex 0A 0B 1D`

Radix Tree: Binary Comparable Keys



Inverted Index

Observation

- The tree indexes that we've discussed so far are useful for "point" and "range" queries:
  - Find all customers in the 30308 zip code.
  - Find all orders between June 2020 and September 2020.
- They are not good at keyword searches:
  - Find all Wikipedia articles that contain the word "Trie"

Wikipedia Example

```sql
CREATE TABLE pages (
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,                           -- Text Search
updated DATETIME
);
```

Wikipedia Example

- If we create an index on the content attribute, what does that do?
- This doesn't help our query.
- Our query is also not correct since it will return any occurrence (not only exact matches)

```
CREATE INDEX idx_rev_content ON revisions (content);
SELECT pageID FROM revisions WHERE content LIKE '%Trie%';
```

Georgia Tech

Inverted Index

- 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.
- Major DBMSs support these natively (*e.g.*, PostgreSQL Generalized Inverted Index (GIN))
- There are also specialized DBMSs (*e.g.*, Lucene, Elasticsearch)

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

Conclusion

- B+Trees are still the way to go for tree indexes.
- Next Class
  - How to make indexes thread-safe!