Carnegie Mellon University

08 Tree Indexes – Part II

Intro to Database Systems
15-445/15-645
Fall 2020

AP
Andy Pavlo
Computer Science
Carnegie Mellon University

ADMINISTRIVIA

Homework #2 is due Sunday Oct 4th

Project #2 will be released tonight:
→ Checkpoint #1: Due Sunday Oct 11th
→ Checkpoint #2: Due Sunday Oct 25th

UPCOMING DATABASE TALKS

CockroachDB Query Optimizer
→ Monday Sept 28th @ 5pm ET

Apache Arrow
→ Monday Oct 5th @ 5pm ET

DataBricks Query Optimizer
→ Monday Oct 12th @ 5pm ET

TODAY'S AGENDA

More B+Trees

Additional Index Magic

Tries / Radix Trees

Inverted Indexes

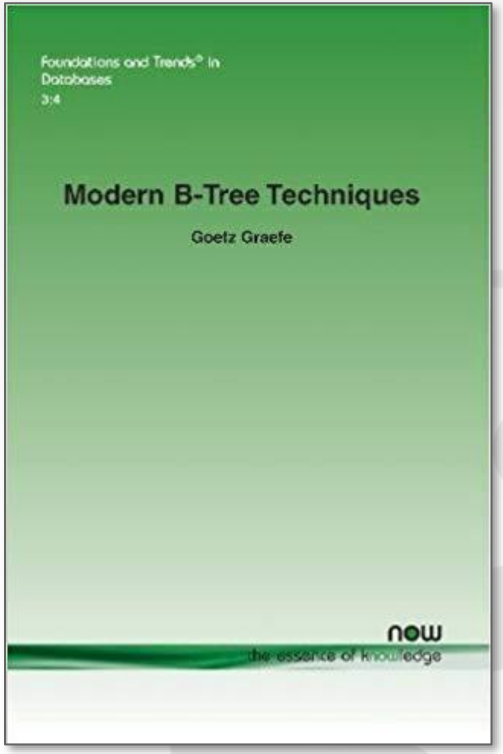
B+TREE DESIGN CHOICES

Node Size

Merge Threshold

Variable Length Keys

Intra-Node Search



NODE SIZE

The slower the storage device, the larger the optimal node size for a B+Tree.

→ HDD ~1MB
→ SSD: ~10KB
→ In-Memory: ~512B

Optimal sizes can vary depending on the workload

→ Leaf Node Scans vs. Root-to-Leaf Traversals

MERGE THRESHOLD

Some DBMSs do not always merge nodes when it is half full.

Delaying a merge operation may reduce the amount of reorganization.

It may also be better to just let underflows to exist and then periodically rebuild entire tree.

VARIABLE LENGTH KEYS

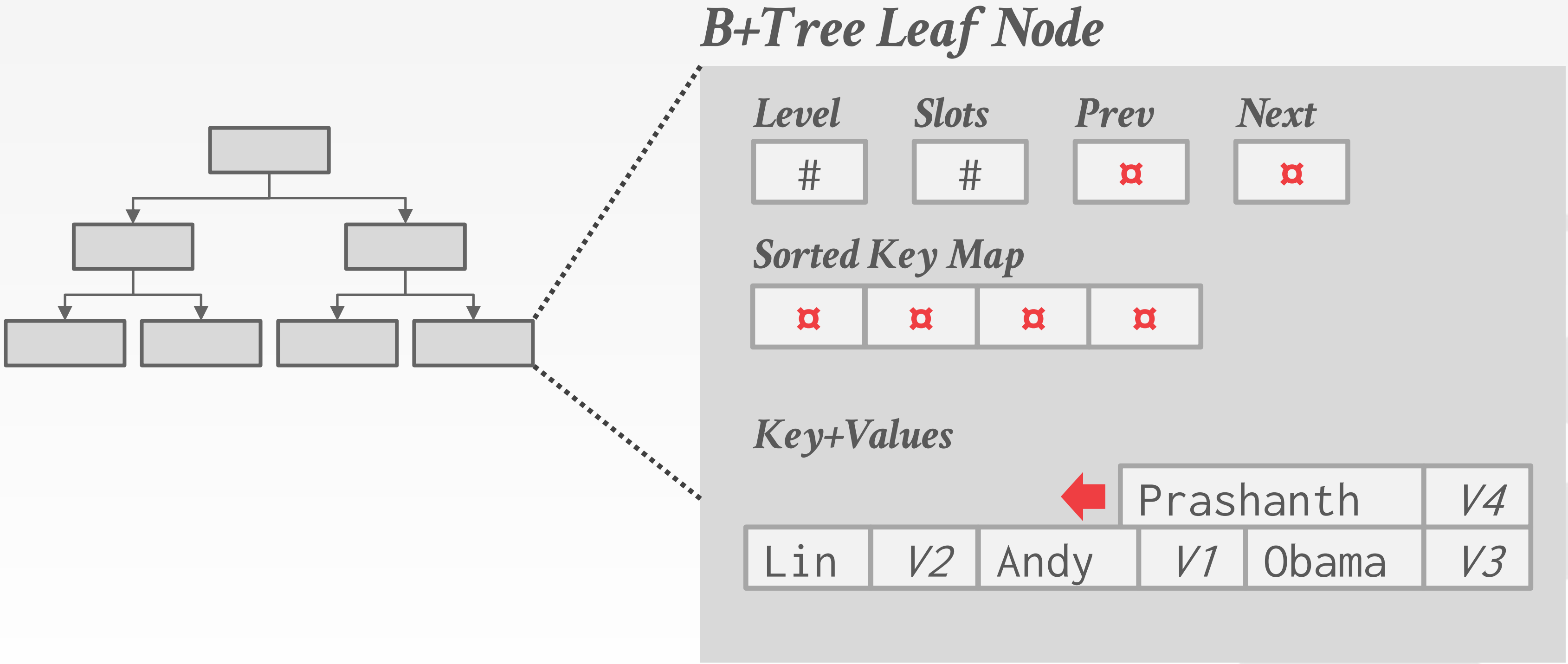
Approach #1: Pointers
→ Store the keys as pointers to the tuple's attribute.

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

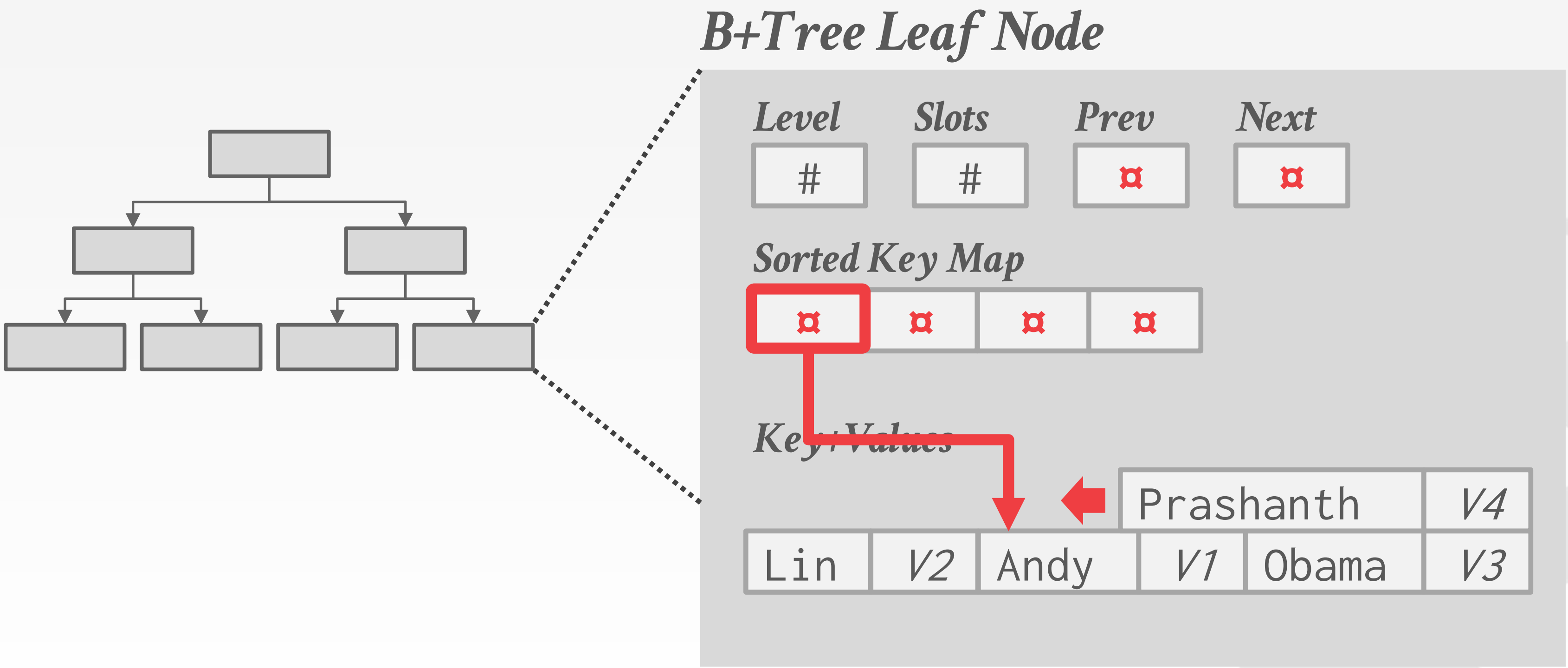
Approach #3: Padding
→ Always pad the key to be max length of the key type.

Approach #4: Key Map / Indirection
→ Embed an array of pointers that map to the key + value list within the node.

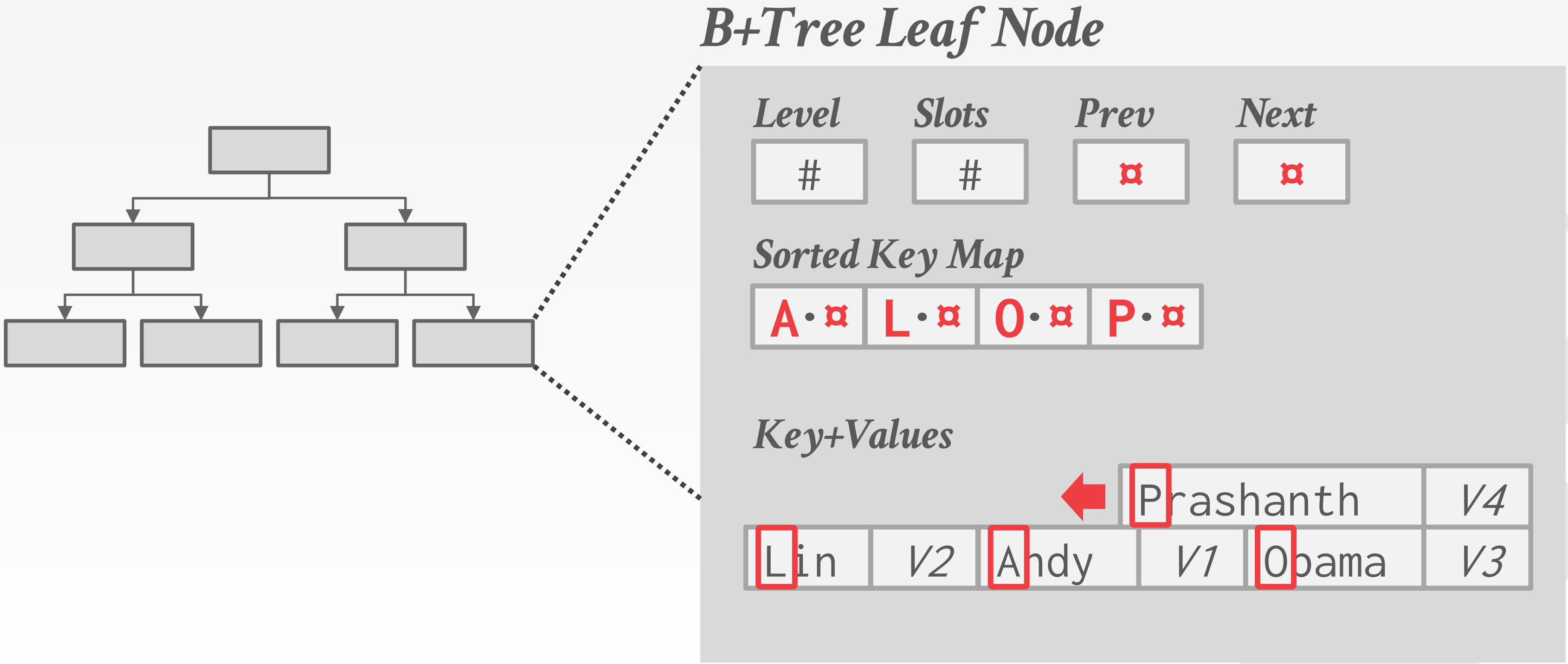
KEY MAP / INDIRECTION



KEY MAP / INDIRECTION



KEY MAP / INDIRECTION



INTRA-NODE SEARCH

Approach #1: Linear

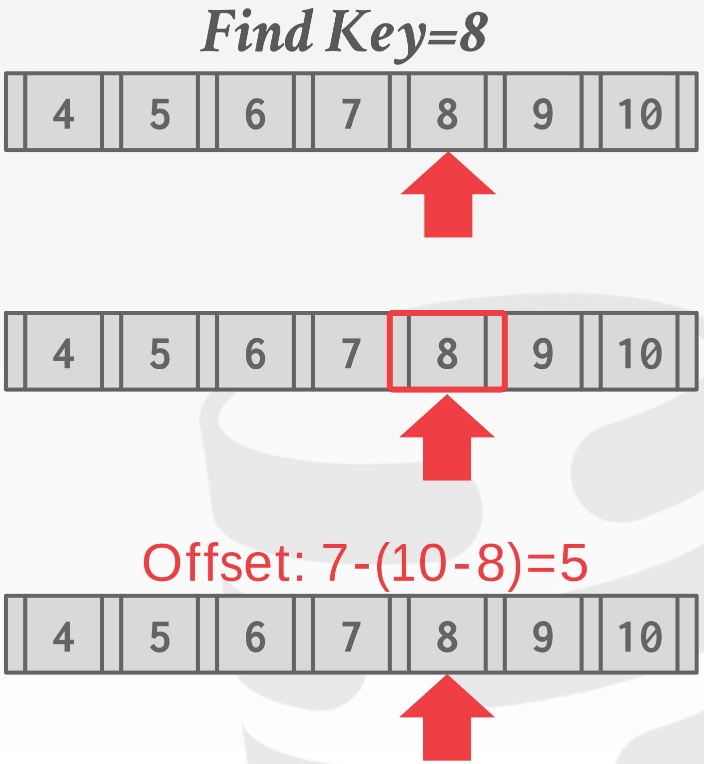
→ Scan node keys from beginning to end.

Approach #2: Binary

→ Jump to middle key, pivot left/right depending on comparison.

Approach #3: Interpolation

→ Approximate location of desired key based on known distribution of keys.



OPTIMIZATIONS

Prefix Compression

Deduplication

Suffix Truncation

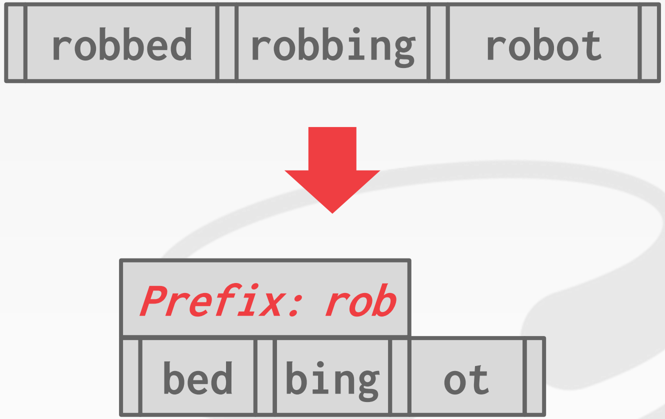
Bulk Insert

Pointer Swizzling

PREFIX COMPRESSION

Sorted keys in the same leaf node are likely to have the same prefix.

Instead of storing the entire key each time, extract common prefix and store only unique suffix for each key.
→ Many variations.

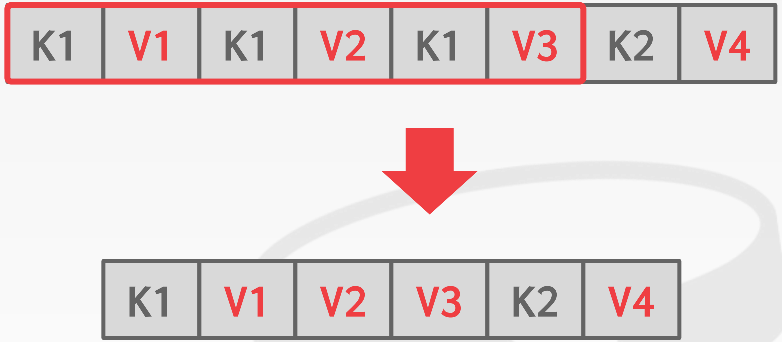


DEDUPLICATION

Non-unique indexes can end up storing keys multiple copies of the same key in leaf nodes.

→ This will make more sense when we talk about MVCC.

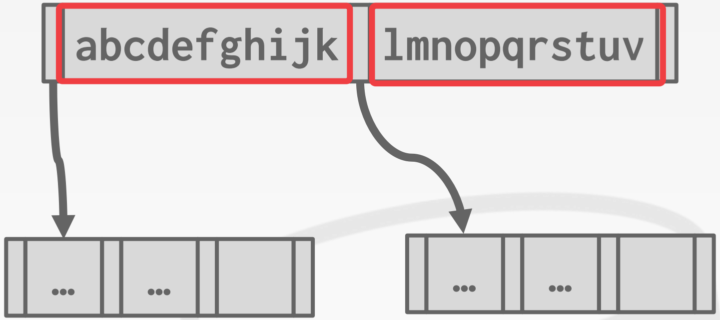
The leaf node can store the key once and then a maintain a list of record ids with that key.



SUFFIX TRUNCATION

The keys in the inner nodes are only used to "direct traffic".
→ We don't need the entire key.

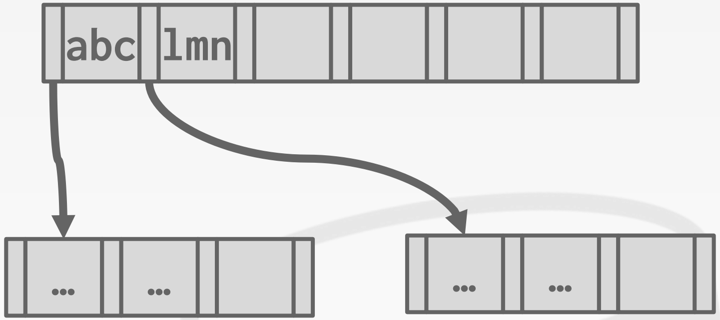
Store a minimum prefix that is needed to correctly route probes into the index.



SUFFIX TRUNCATION

The keys in the inner nodes are only used to "direct traffic".
→ We don't need the entire key.

Store a minimum prefix that is needed to correctly route probes into the index.

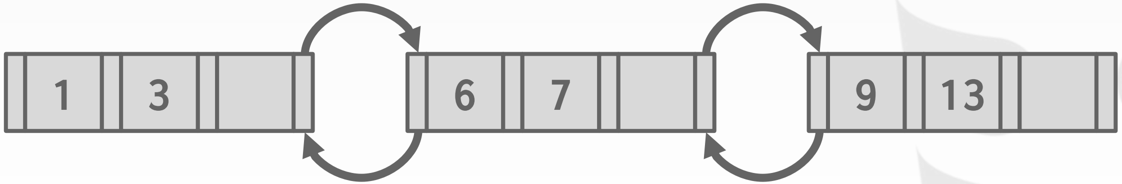


BULK INSERT

The fastest way to build a new B+Tree for an existing table is to first sort the keys and then build the index from the bottom up.

Keys: 3, 7, 9, 13, 6, 1

Sorted Keys: 1, 3, 6, 7, 9, 13

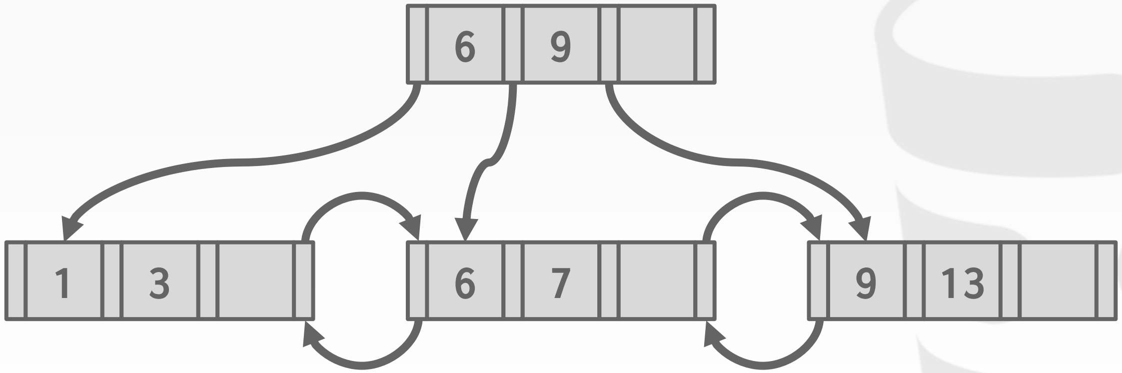


BULK INSERT

The fastest way to build a new B+Tree for an existing table is to first sort the keys and then build the index from the bottom up.

Keys: 3, 7, 9, 13, 6, 1

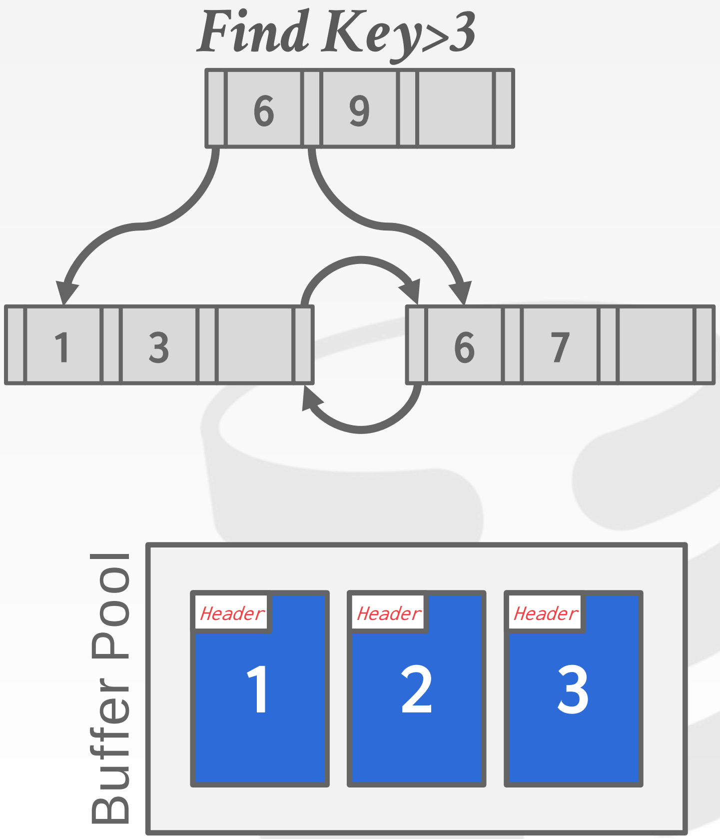
Sorted Keys: 1, 3, 6, 7, 9, 13



POINTER SWIZZLING

Nodes use page ids to reference other nodes in the index. The DBMS must get the memory location from the page table during traversal.

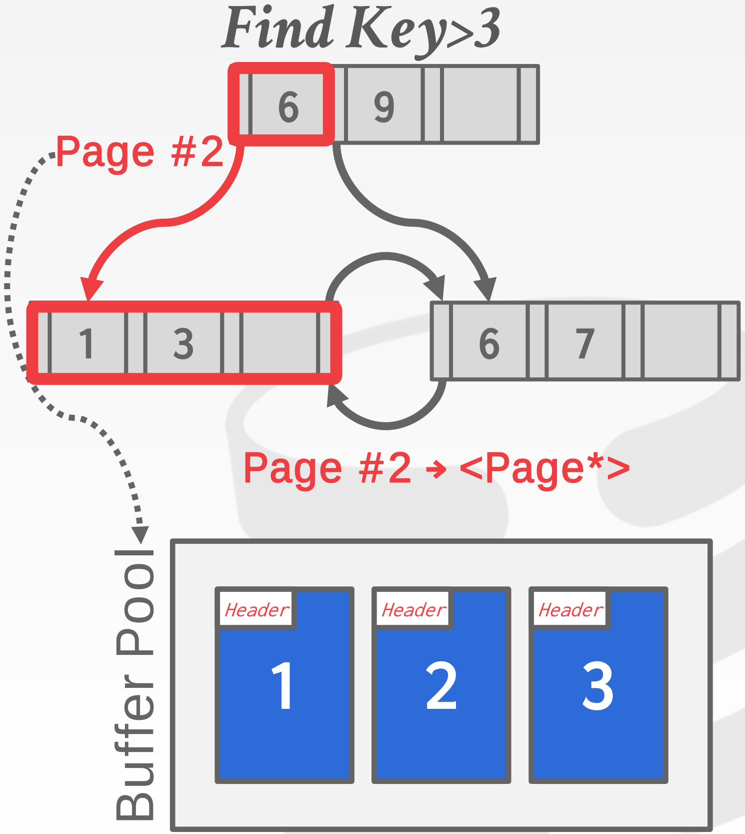
If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids. This avoids address lookups from the page table.



POINTER SWIZZLING

Nodes use page ids to reference other nodes in the index. The DBMS must get the memory location from the page table during traversal.

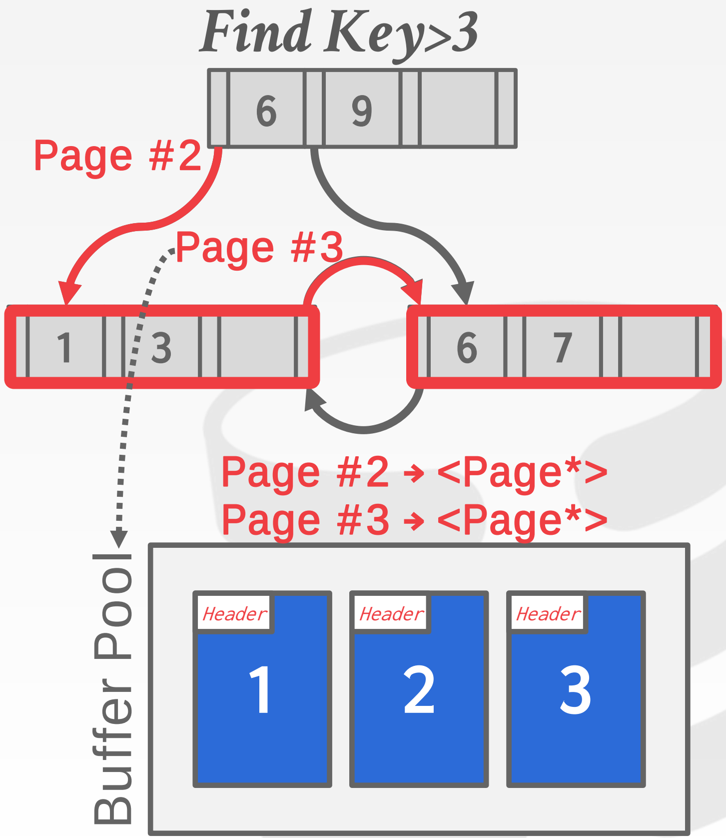
If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids. This avoids address lookups from the page table.



POINTER SWIZZLING

Nodes use page ids to reference other nodes in the index. The DBMS must get the memory location from the page table during traversal.

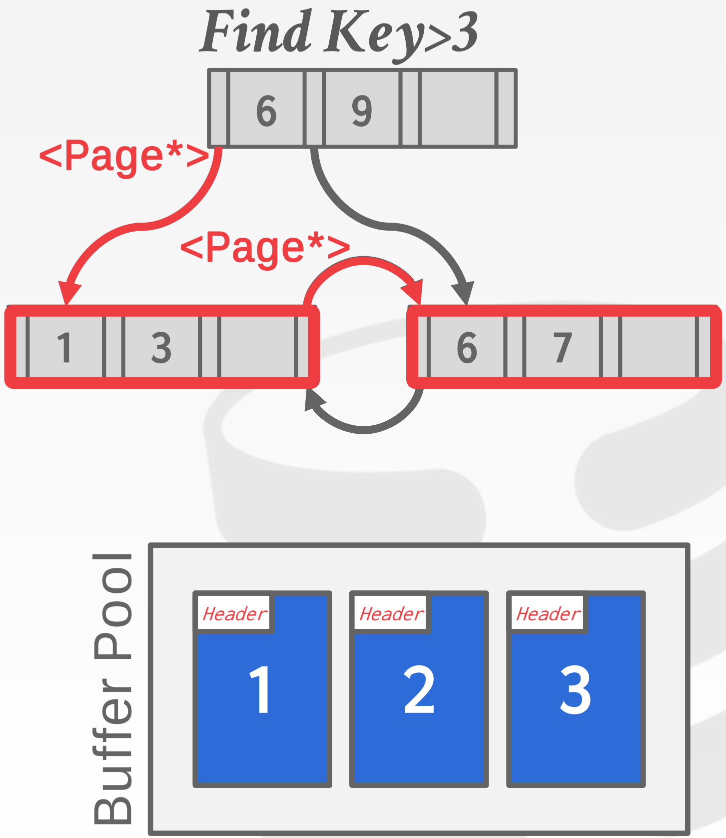
If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids. This avoids address lookups from the page table.



POINTER SWIZZLING

Nodes use page ids to reference other nodes in the index. The DBMS must get the memory location from the page table during traversal.

If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids. This avoids address lookups from the page table.



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';
```

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
```

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.

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

Also called **index-only scans**.

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

```
SELECT b FROM foo
 WHERE a = 123;
```

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.

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

Also called **index-only scans**.

```
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.

```
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.

```
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.

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);
```

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.

```
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));
```

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.

```
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));
```

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.

```
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 foo (login)
 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.

```
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 foo (login)
 WHERE EXTRACT(dow FROM login) = 2;
```

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.

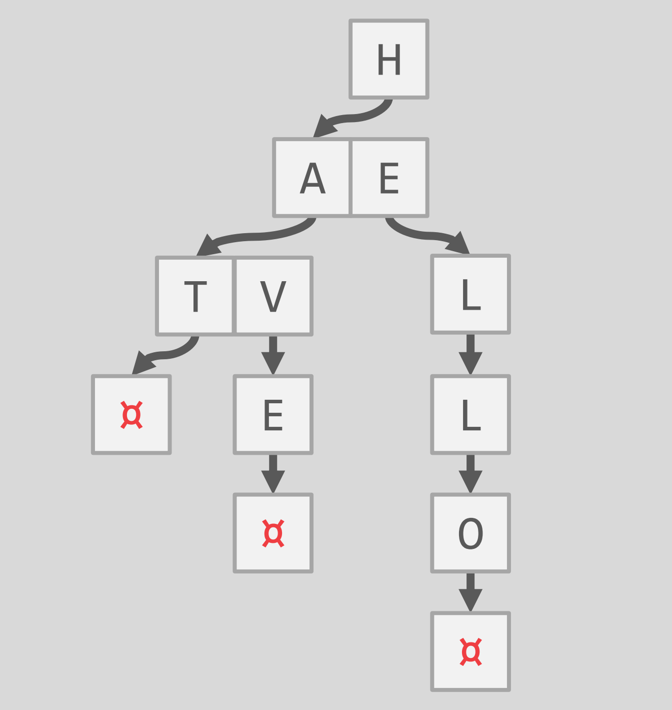
This means that you could have (at least) one buffer pool page miss per level in the tree just to find out a key does not exist.

TRIE INDEX

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

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

Keys: HELLO, HAT, HAVE

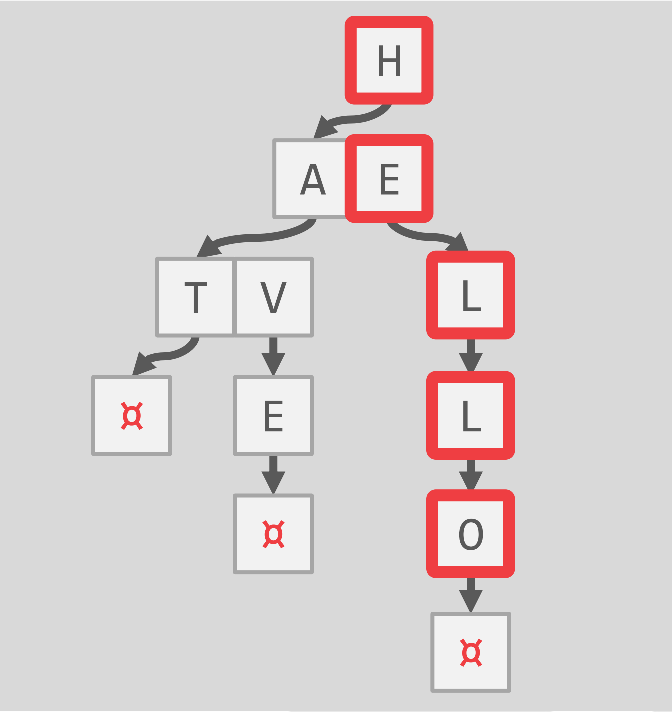


TRIE INDEX

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***.

Keys: HELLO HAT, HAVE



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.

TRIE KEY SPAN

The **span** of a trie level is the number of bits that each partial key / digit represents.

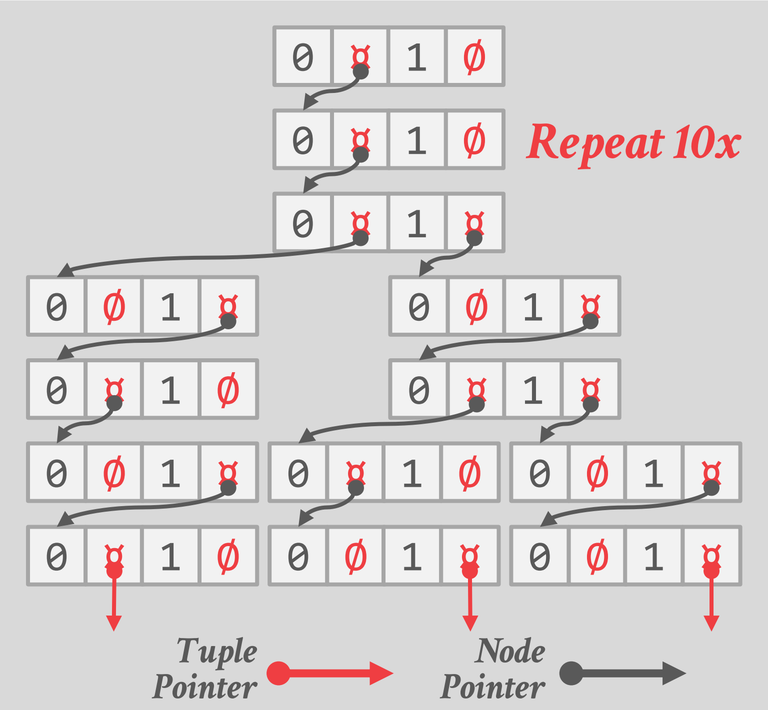
→ If the digit exists in the corpus, then store a pointer to the next level in the trie branch. Otherwise, store null.

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

→ ***n***-way Trie = Fan-Out of ***n***

TRIE KEY SPAN

1-bit Span Trie



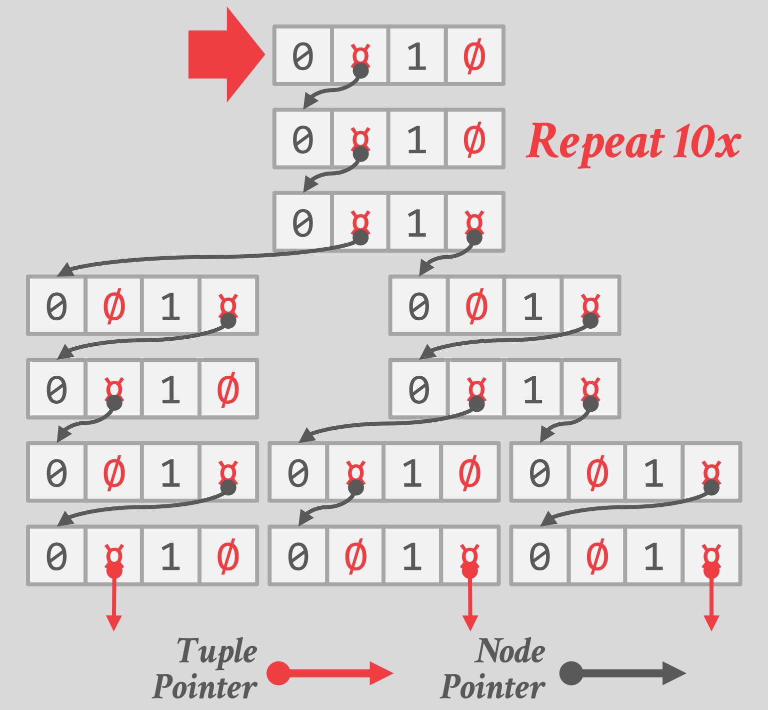
K10→ 00000000 00001010

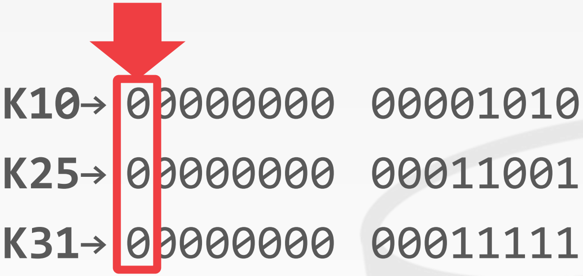
K25→ 00000000 00011001

K31→ 00000000 00011111

TRIE KEY SPAN

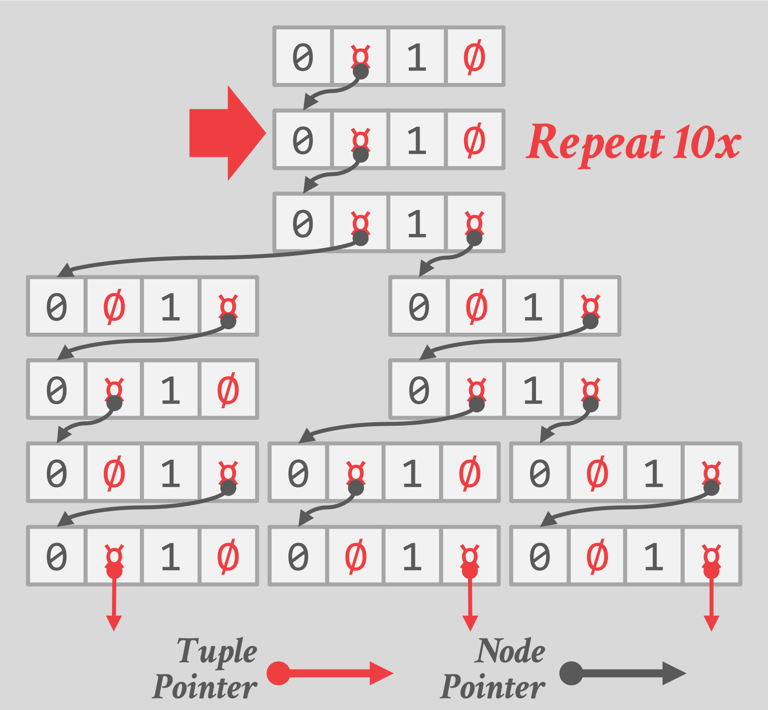
1-bit Span Trie





TRIE KEY SPAN

1-bit Span Trie



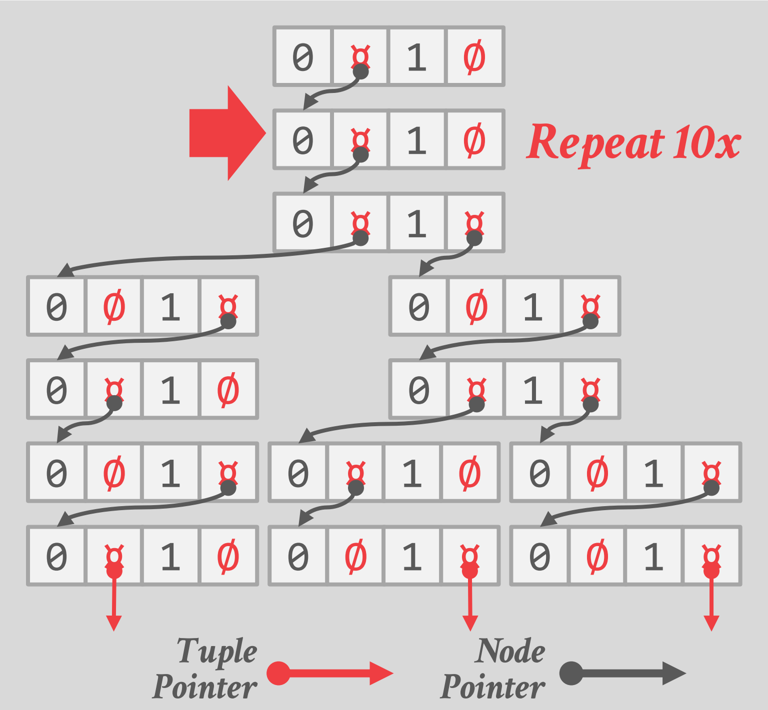
K10→ 00000000 00001010

K25→ 00000000 00011001

K31→ 00000000 00011111

TRIE KEY SPAN

1-bit Span Trie



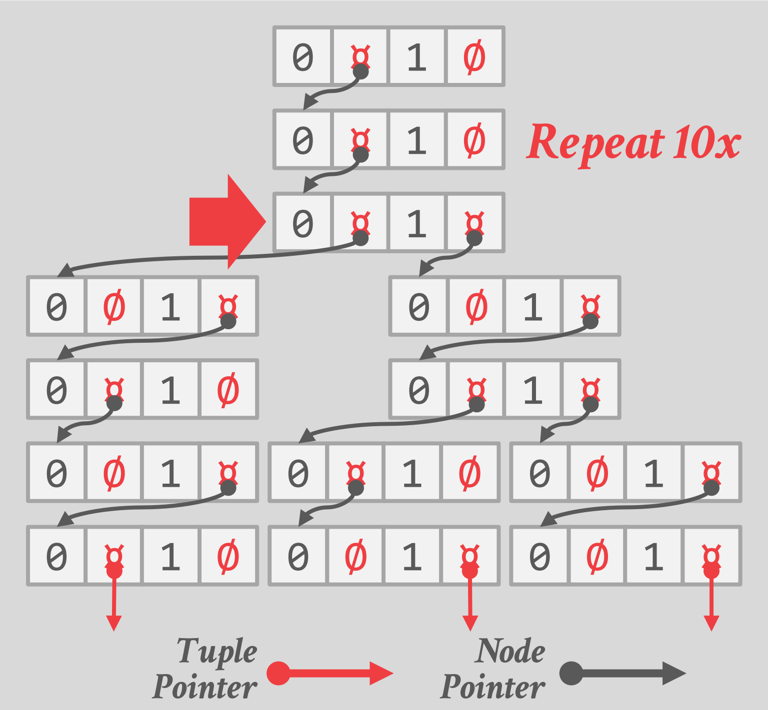
K10→ 00000000 00001010

K25→ 00000000 00011001

K31→ 00000000 00011111

TRIE KEY SPAN

1-bit Span Trie



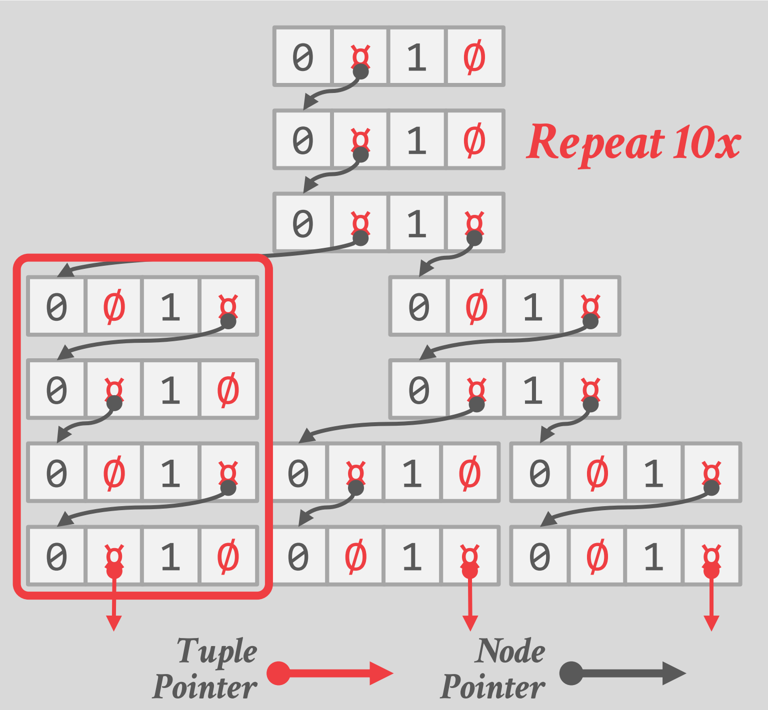
K10→ 00000000 00001010

K25→ 00000000 00011001

K31→ 00000000 00011111

TRIE KEY SPAN

1-bit Span Trie



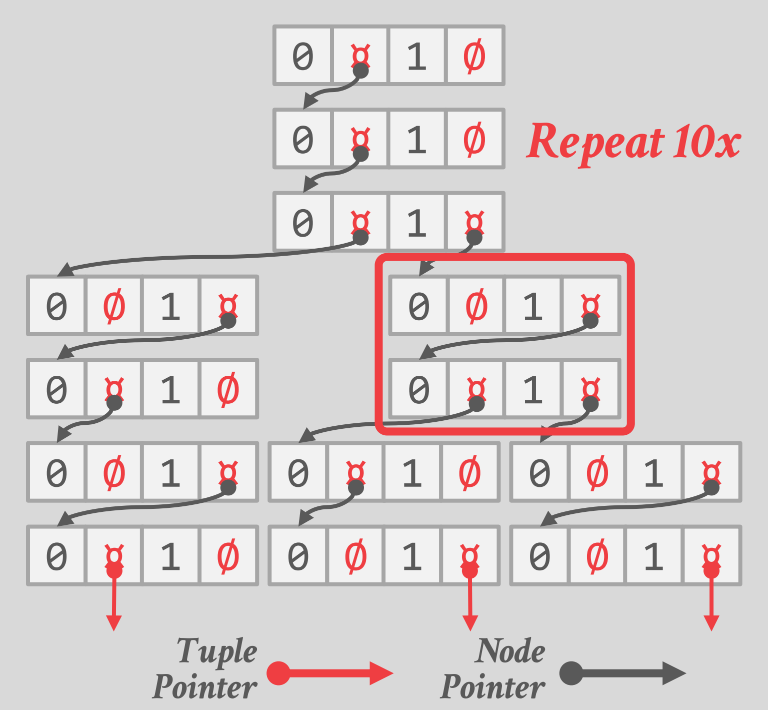
K10→ 00000000 0000$\boxed{1010}$

K25→ 00000000 00011001

K31→ 00000000 00011111

TRIE KEY SPAN

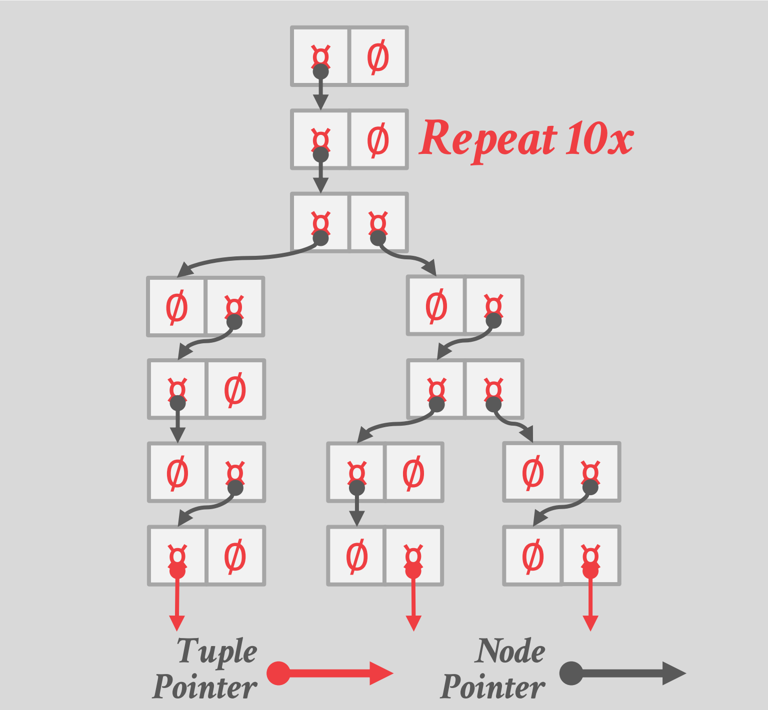
1-bit Span Trie



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

TRIE KEY SPAN

1-bit Span Trie



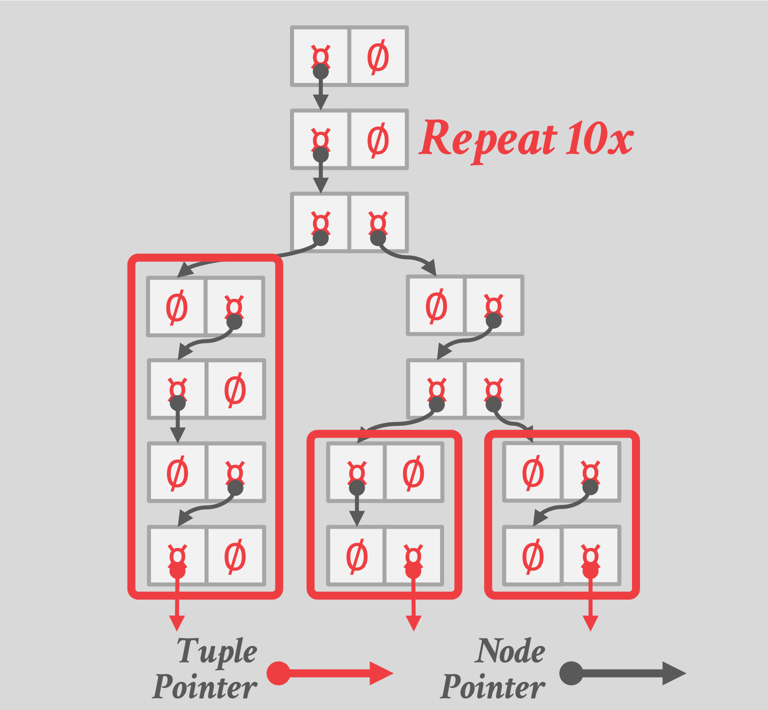
K10→ 00000000 00001010

K25→ 00000000 00011001

K31→ 00000000 00011111

TRIE KEY SPAN

1-bit Span Trie



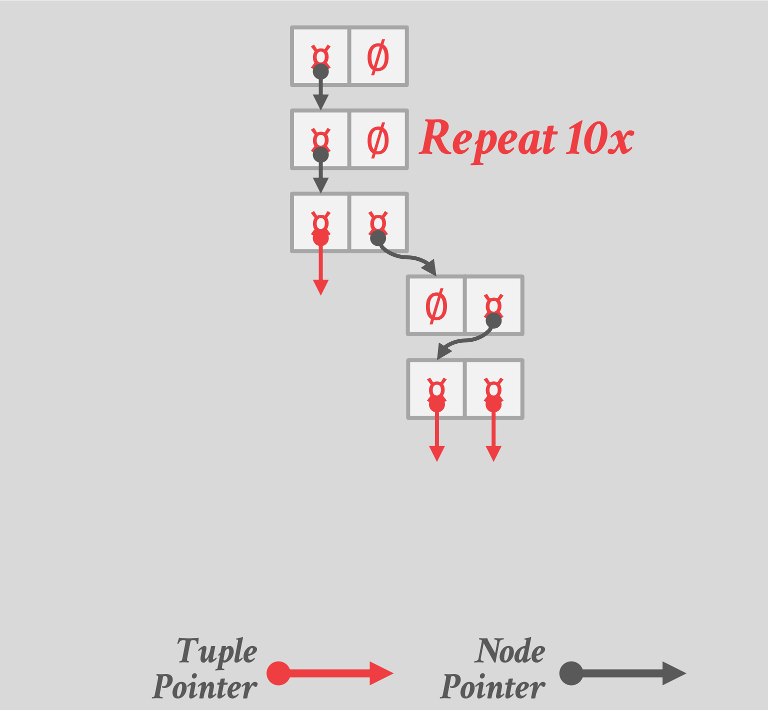
K10→ 00000000 00001010

K25→ 00000000 00011001

K31→ 00000000 00011111

RADIX TREE

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.

OBSERVATION

The tree indexes that we've discussed so far are useful for "point" and "range" queries:

→ Find all customers in the 15217 zipcode.
→ 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"

WIKIPEDIA EXAMPLE

```
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 do?

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

This doesn't help our query.

Our SQL is also not correct...

```
SELECT pageID FROM revisions
 WHERE content LIKE '%Pavlo%';
```

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.

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!