Carnegie Mellon University

08 Tree Indexes –Part II

Intro to Database Systems
15-445/15-645
Fall 2019

AP
Andy Pavlo
Computer Science
Carnegie Mellon University

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

page_id + offset (a unique tuple)

→ Add the tuple's unique record id as part of the key to ensure that all keys are unique.

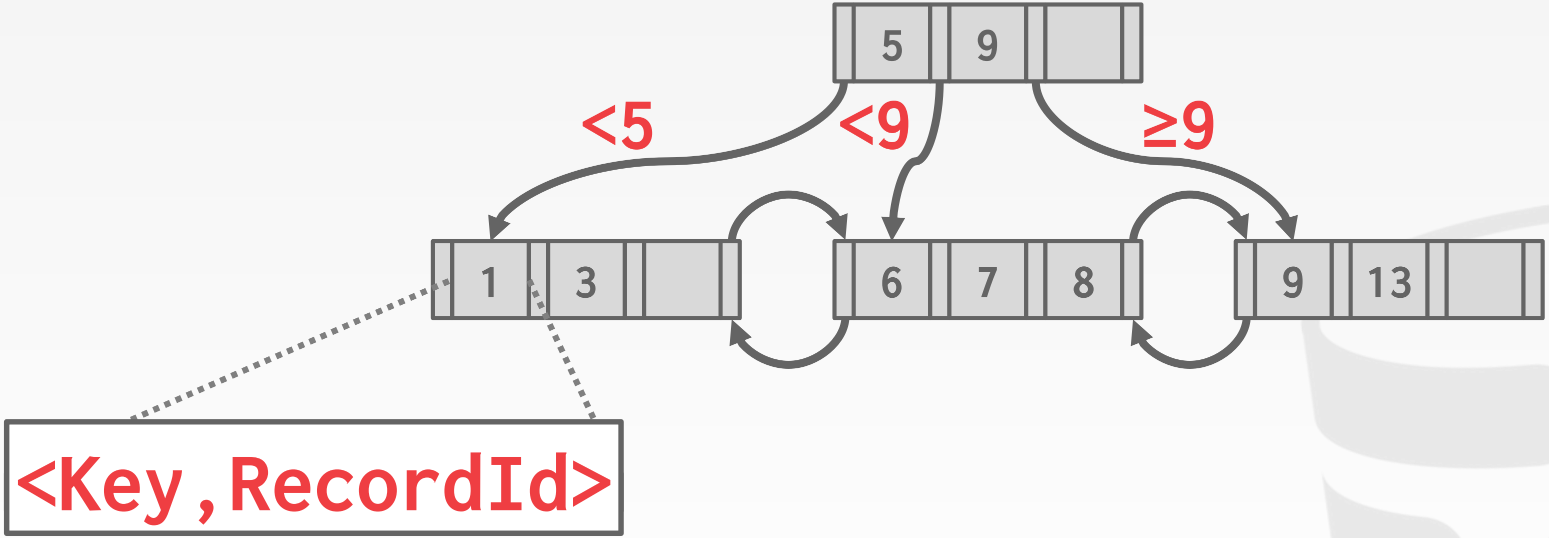
→ The DBMS can still use partial keys to find tuples.

Approach #2: Overflow Leaf Nodes

→ Allow leaf nodes to spill into overflow nodes that contain the duplicate keys.

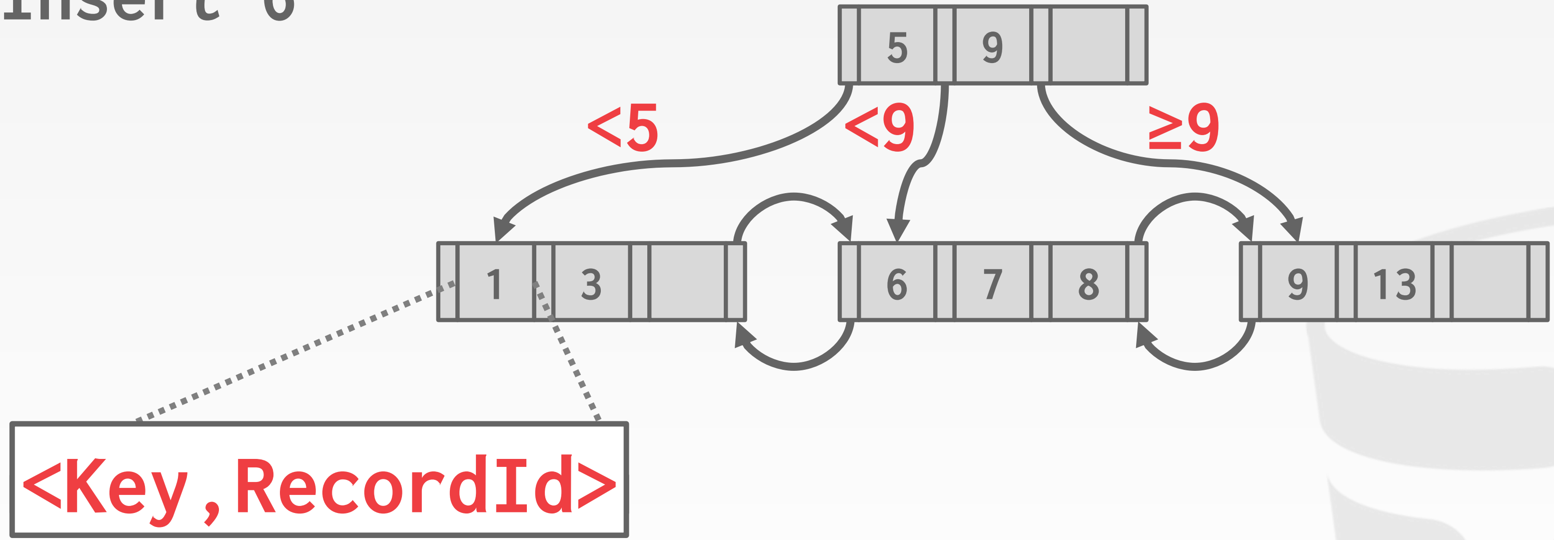
→ This is more complex to maintain and modify.

B+TREE: APPEND RECORD ID



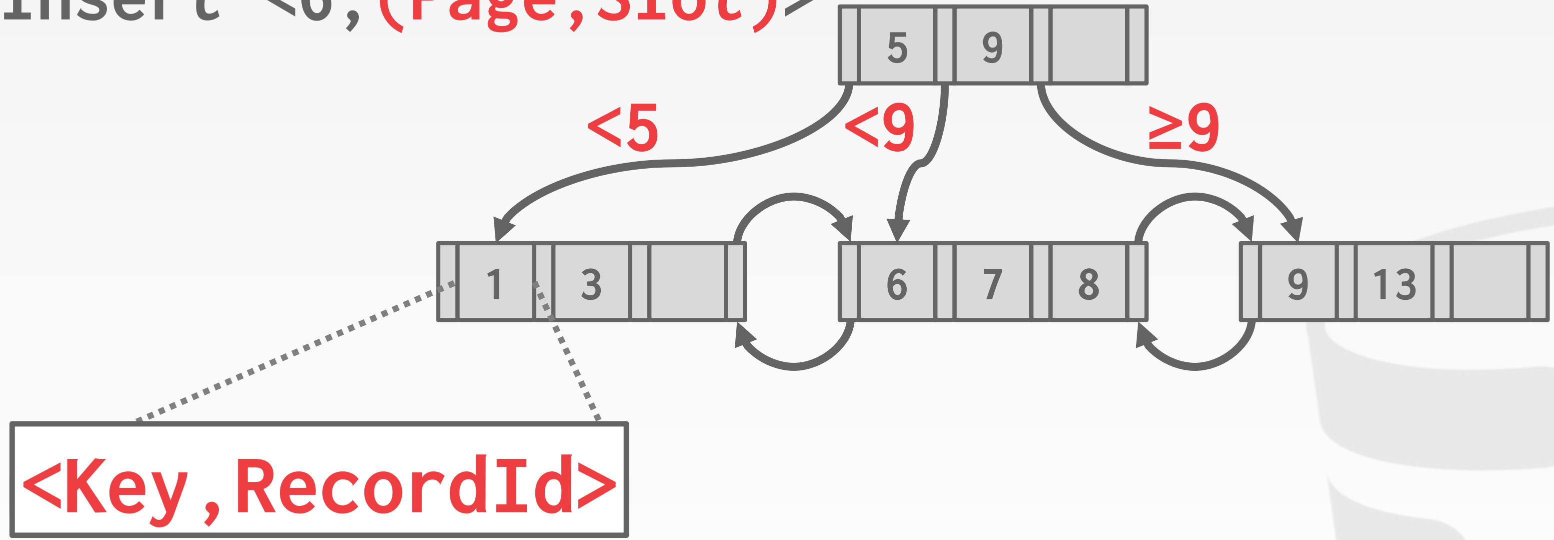
B+TREE: APPEND RECORD ID

Insert 6



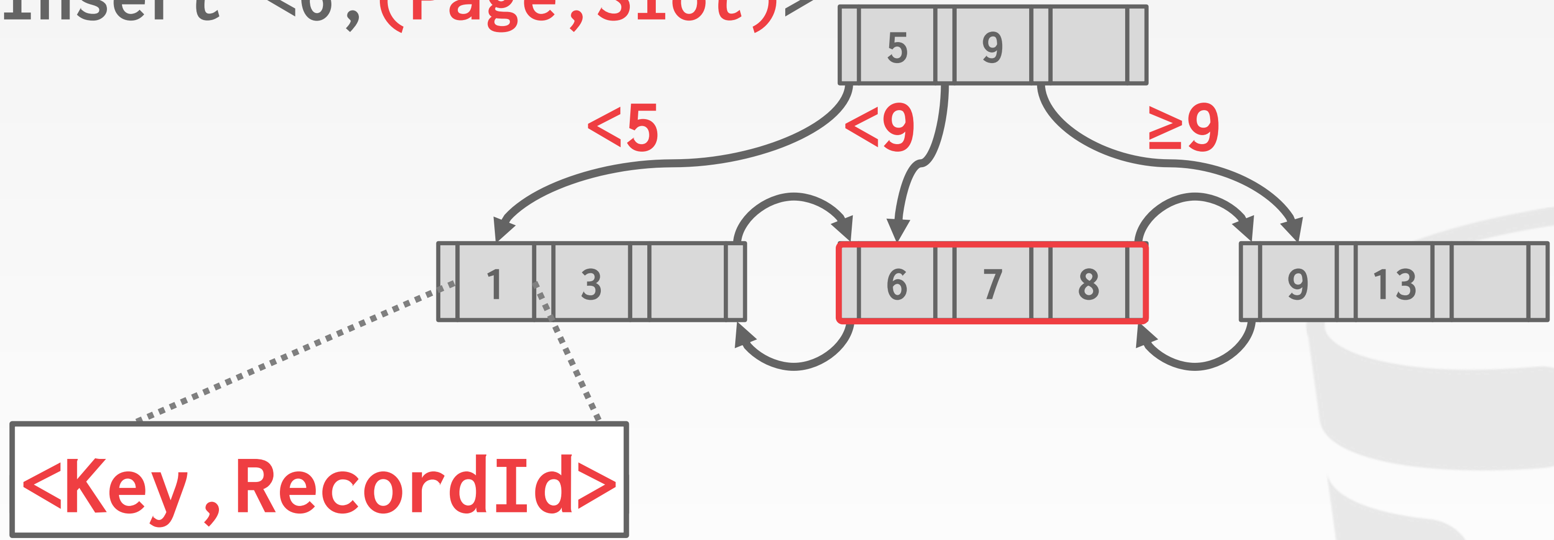
B+TREE: APPEND RECORD ID

Insert <6,(Page,Slot)>



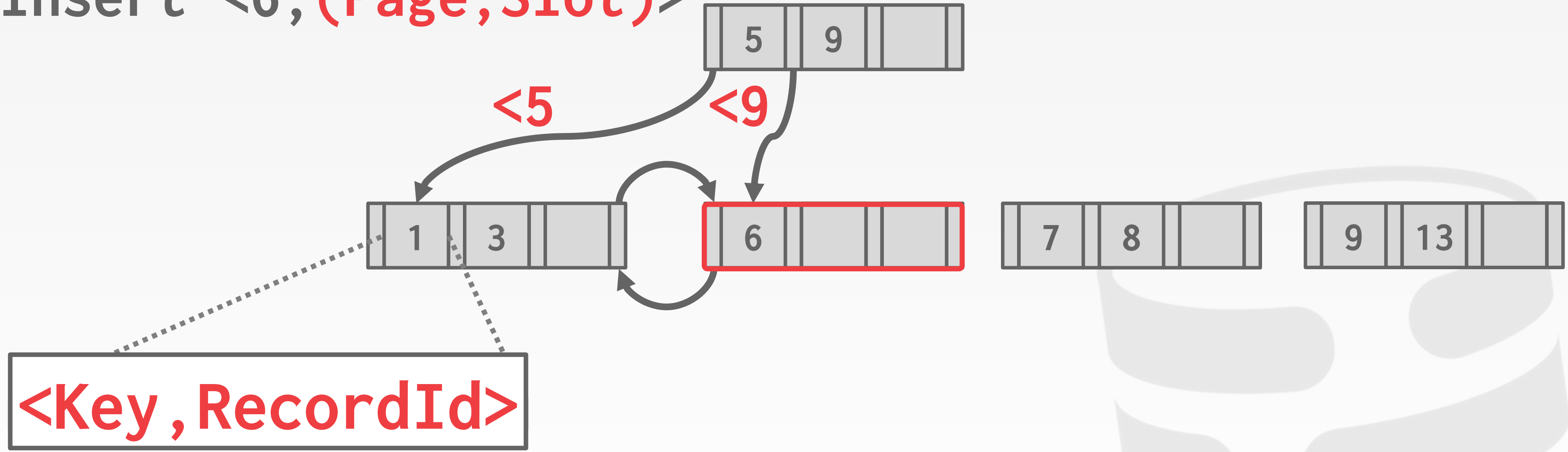
B+TREE: APPEND RECORD ID

Insert <6,(Page,Slot)>



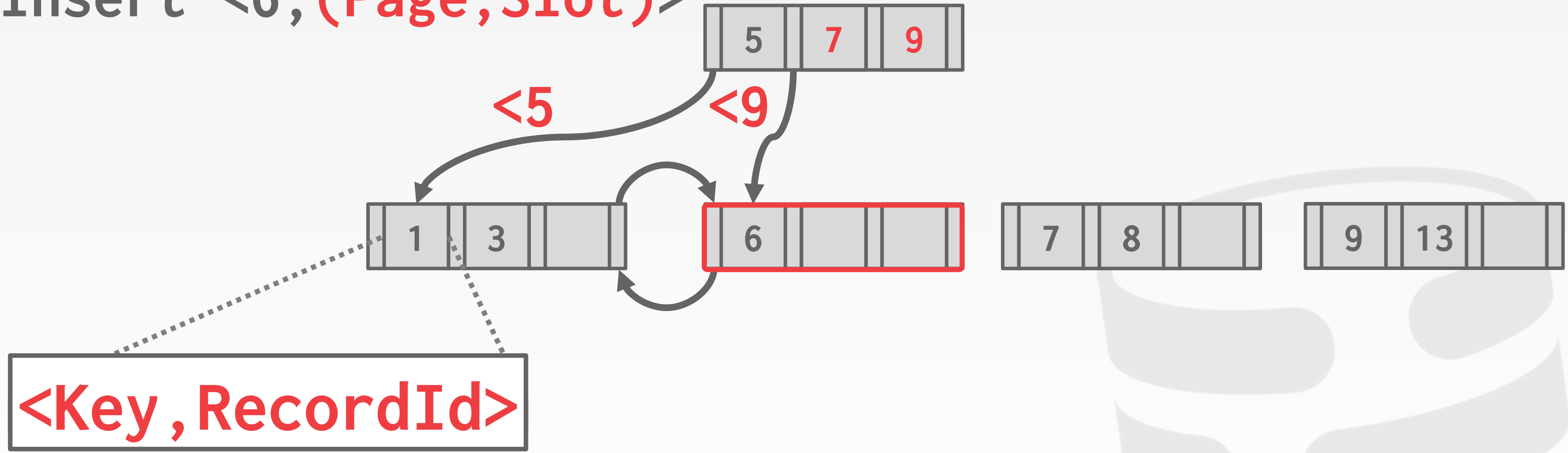
B+TREE: APPEND RECORD ID

Insert <6,(Page,Slot)>



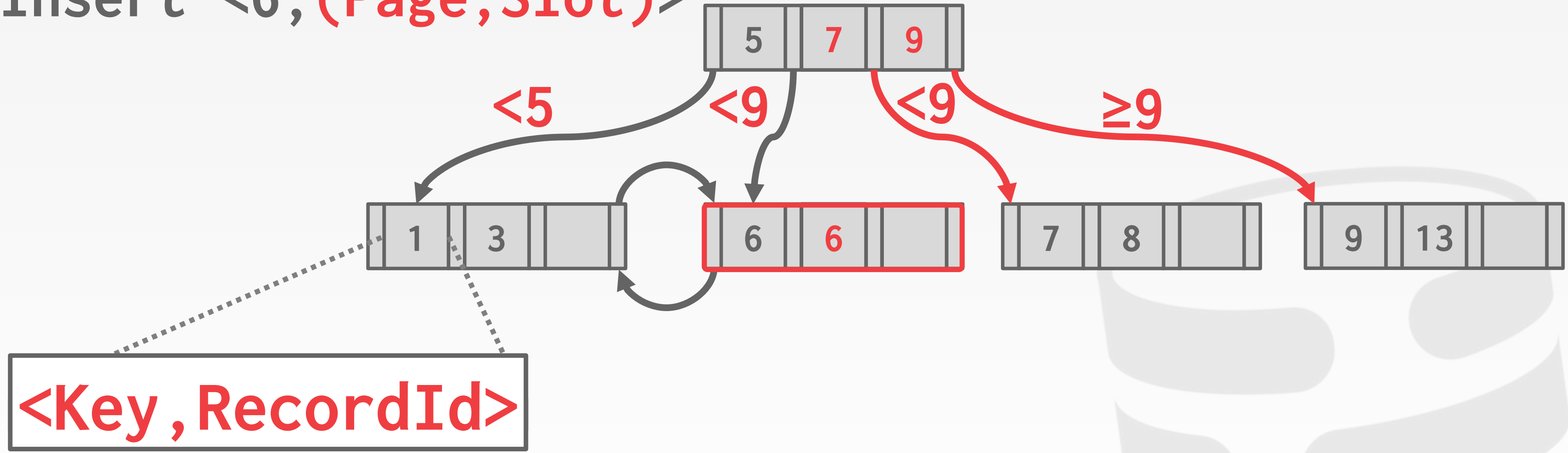
B+TREE: APPEND RECORD ID

Insert <6,(Page,Slot)>



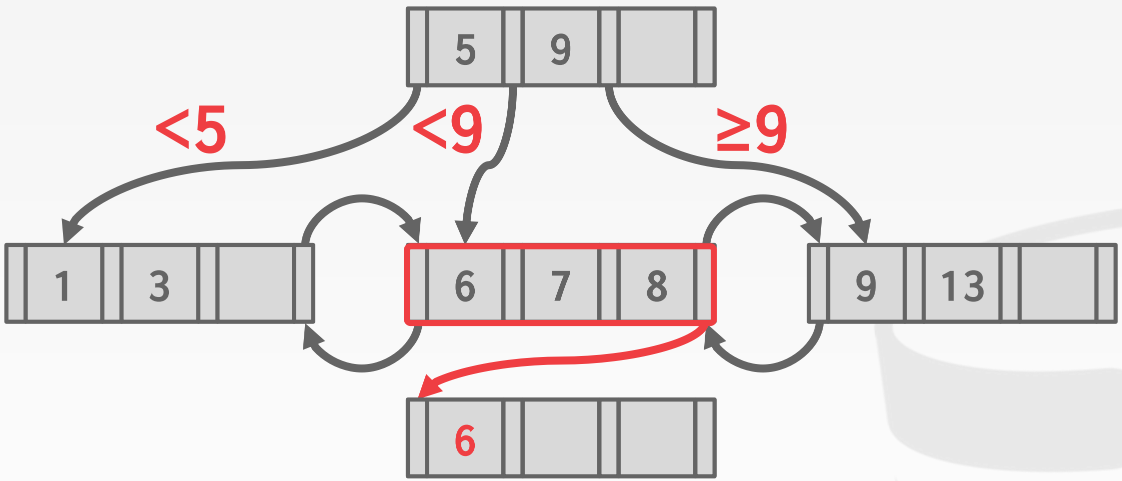
B+TREE: APPEND RECORD ID

Insert <6,(Page,Slot)>



B+TREE: OVERFLOW LEAF NODES

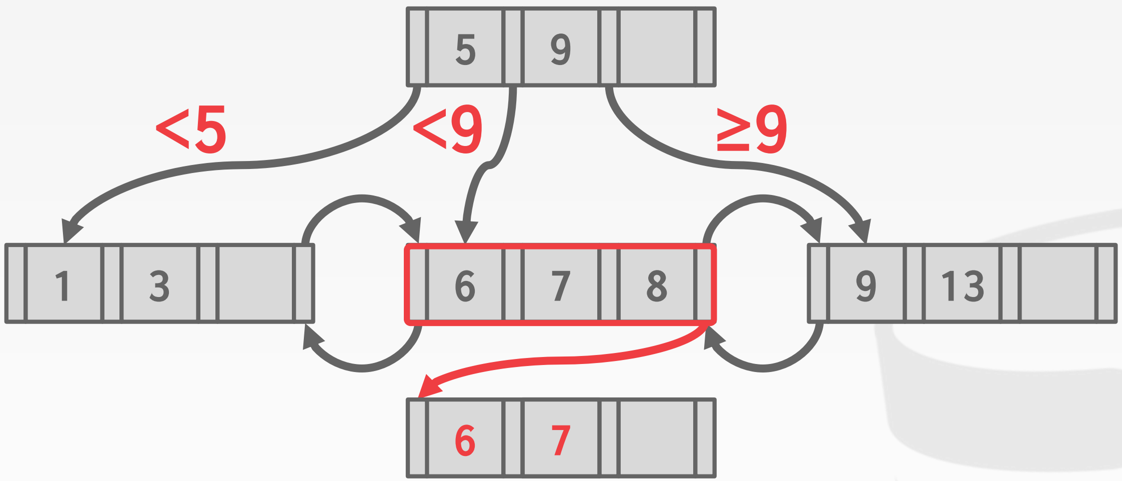
Insert 6



B+TREE: OVERFLOW LEAF NODES

Insert 6

Insert 7

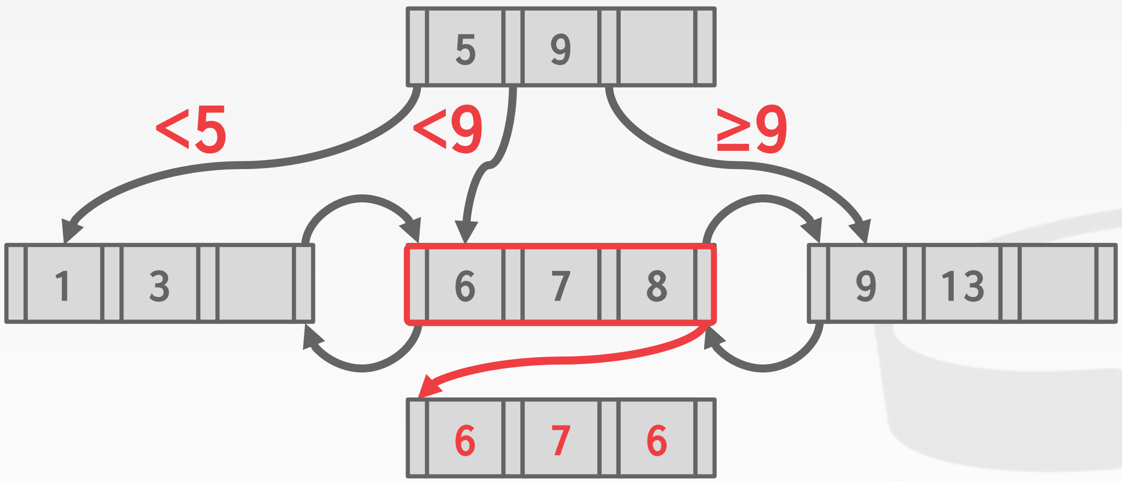


B+TREE: OVERFLOW LEAF NODES

Insert 6

Insert 7

Insert 6



DEMO

B+Tree vs. Hash Indexes

Table Clustering

Sequential scan is always the default operation or access method for the database system. If they can’t find what it wants using index, it always default using a sequential scan.

```
 Bitmap Heap Scan on emails  (cost=3735.55..192161.72 rows=137748 width=278)
   Recheck Cond: ((email)::text = '00@00.000'::text)
   ->  Bitmap Index Scan on idx_emails_hash  (cost=0.00..3701.11 rows=137748 width=0)
         Index Cond: ((email)::text = '00@00.000'::text)
(4 rows)

Time: 1.110 ms
pavlo=# SELECT * FROM emails WHERE email = '00@00.000';
    id    |   email
----------+-----------
 13304310 | 00@00.000
(1 row)

Time: 1.057 ms
pavlo=# EXPLAIN SELECT * FROM emails WHERE email LIKE '00@00%';
                          QUERY PLAN
----------------------------------------------------------------
 Seq Scan on emails  (cost=0.00..539620.70 rows=2755 width=26)
   Filter: ((email)::text ~~ '00@00%'::text)
(2 rows)

Time: 1.704 ms
```

IMPLICIT INDEXES

Most DBMSs automatically create an index to enforce integrity constraints but <u>not</u> 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)
);
```

IMPLICIT INDEXES

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

→ Primary Keys

→ Unique Constraints

```
CREATE INDEX foo_val1_key
                ON foo (val1);
```

```
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 but <u>not</u> referential constraints (foreign keys).

→ Primary Keys

→ Unique Constraints

```
CREATE INDEX foo_val1_key
          ON foo (val1);
```

```
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 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';
```

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.

All the fields that are necessary to answer the require result for the query can be found in the index itself

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;
```

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.

```
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 <u>not</u> 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';
```

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 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;
```

OBSERVATION

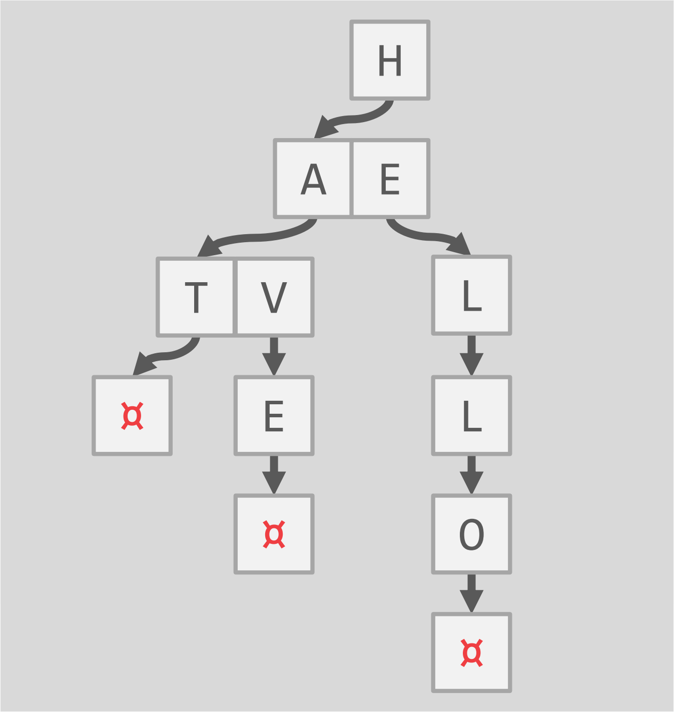
The inner node keys in a B+Tree cannot tell you whether a key exists in the index. You must always traverse to the leaf node.

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.

The inner nodes may have copies of keys that no longer exist, so every time when we wants to determine whether a key exists, I must always traverse to the leaf node.

TRIE INDEX

Keys: HELLO, HAT, HAVE



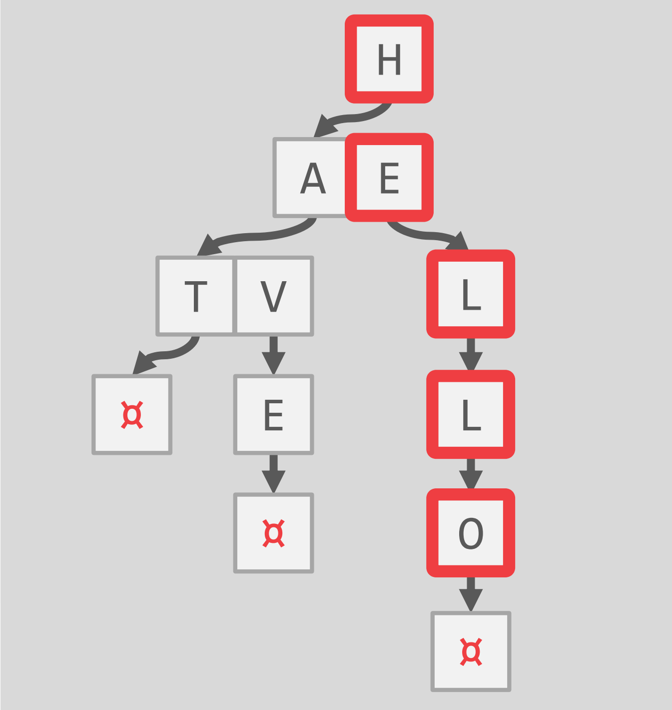
Use a digital representation of keys to examine prefixes one-by-one instead of comparing entire key.

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

Store some atomic subset of our key

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.

TRIE KEY SPAN

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

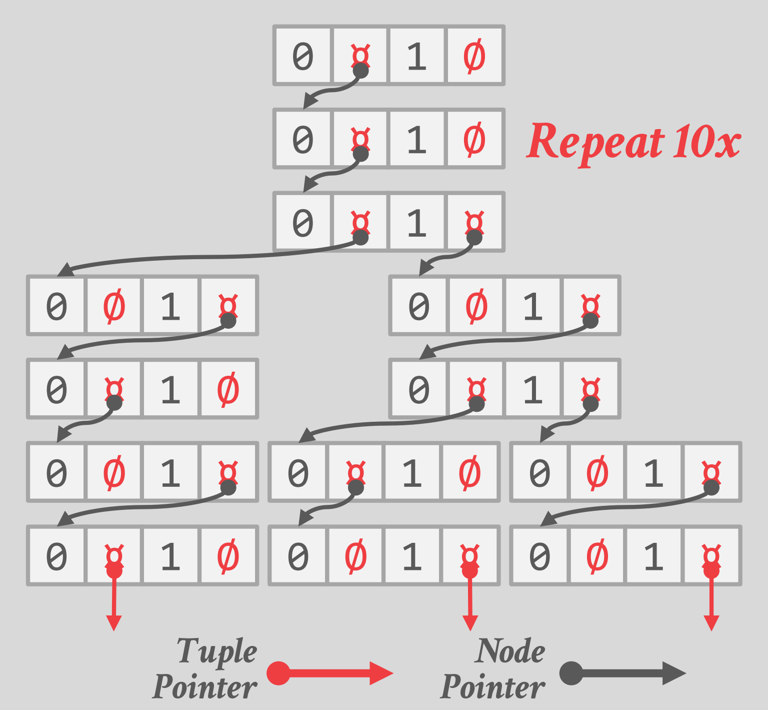
→ If the digit exists in the corpus, then store a pointer to the next level in the trie branch. Otherwise, store null.

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



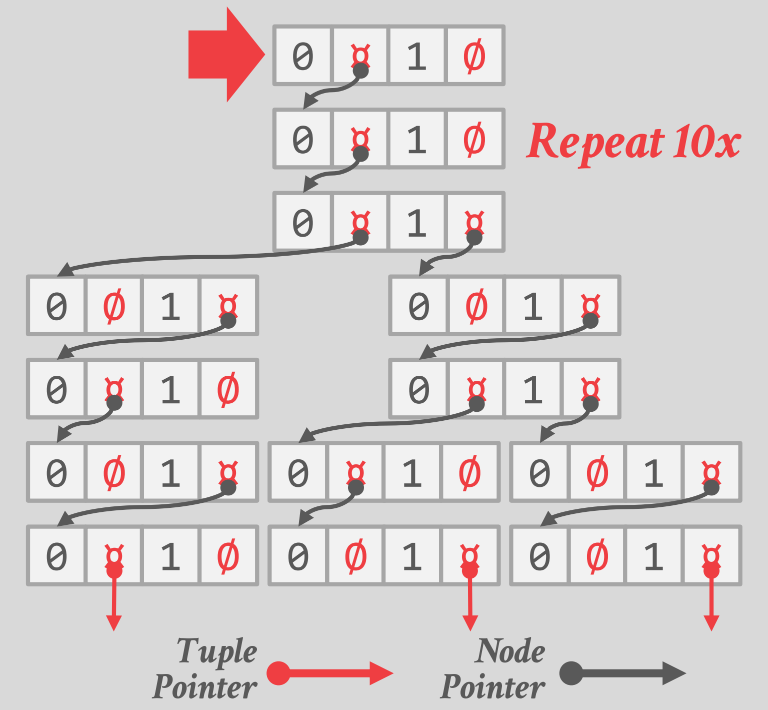
K10→ 00000000 00001010

K25→ 00000000 00011001

K31→ 00000000 00011111

TRIE KEY SPAN

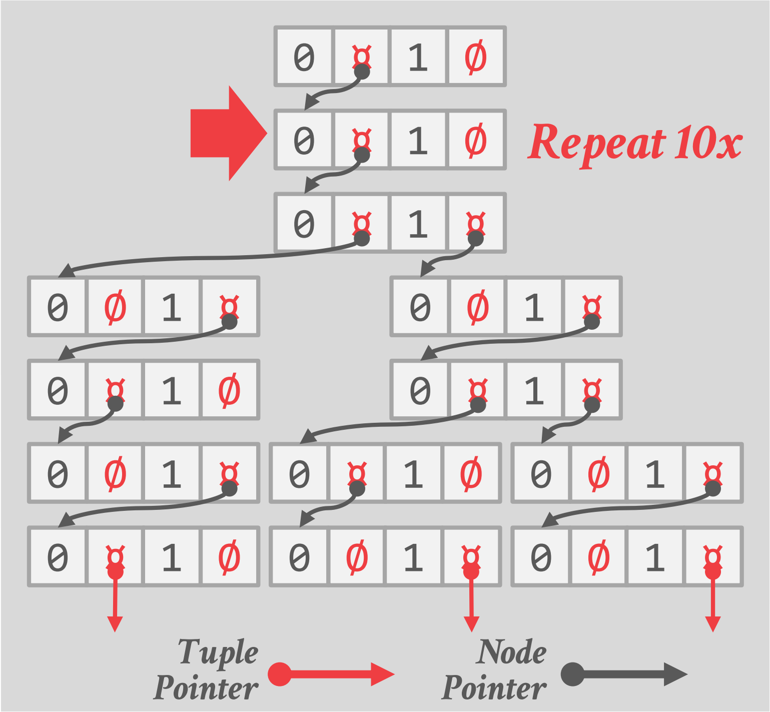
1-bit Span Trie



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

TRIE KEY SPAN

1-bit Span Trie



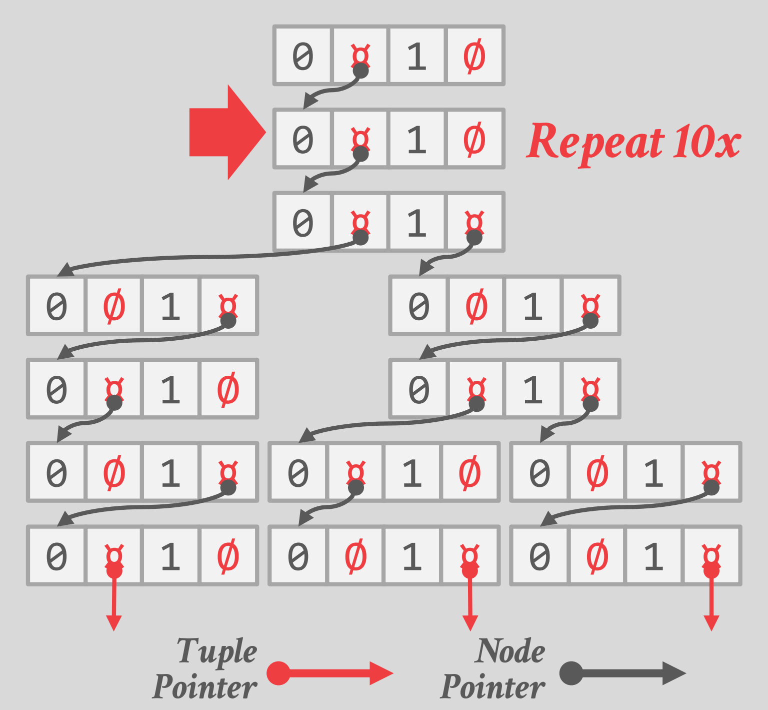
K10→ 00000000 00001010

K25→ 00000000 00011001

K31→ 00000000 00011111

TRIE KEY SPAN

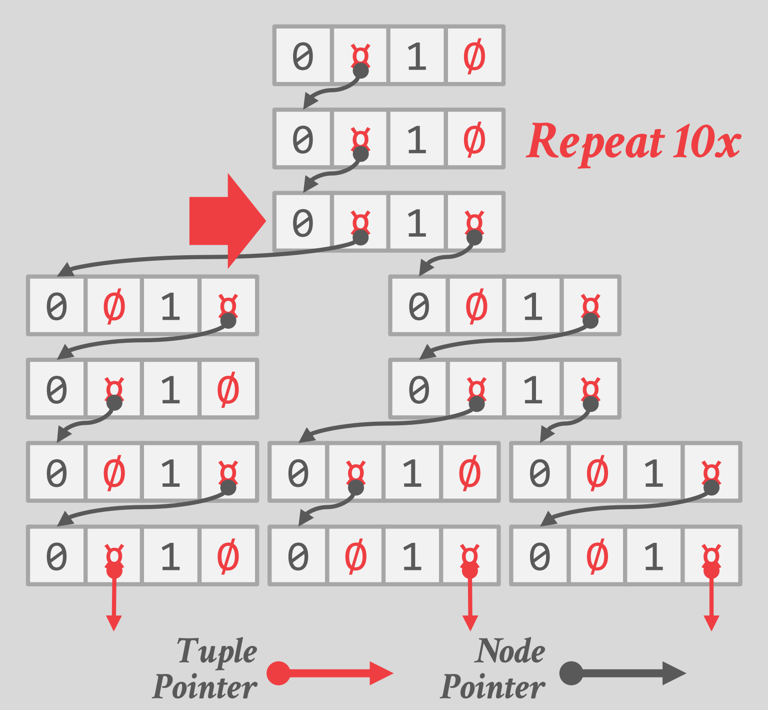
1-bit Span Trie



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

TRIE KEY SPAN

1-bit Span Trie



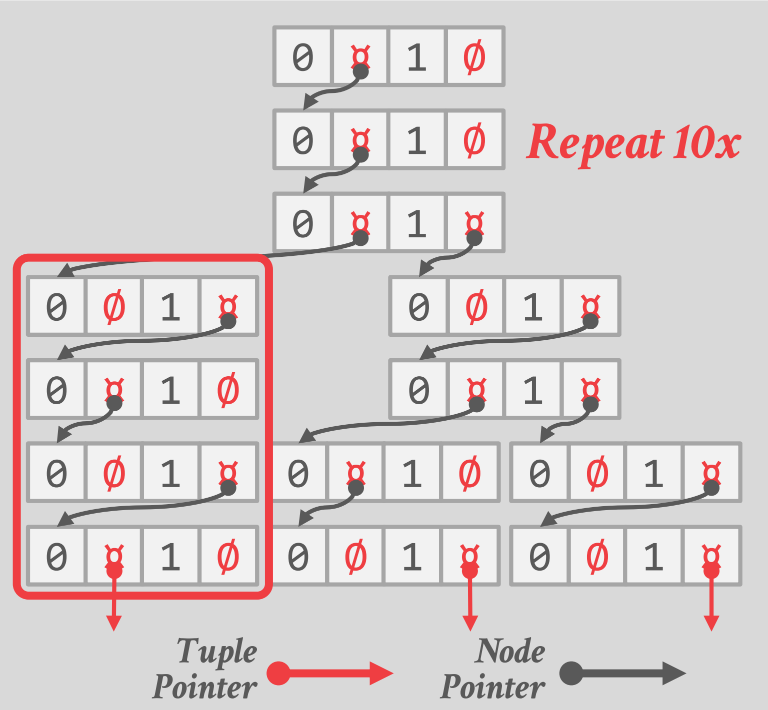
K10→ 00000000 00001010

K25→ 00000000 00011001

K31→ 00000000 00011111

TRIE KEY SPAN

1-bit Span Trie



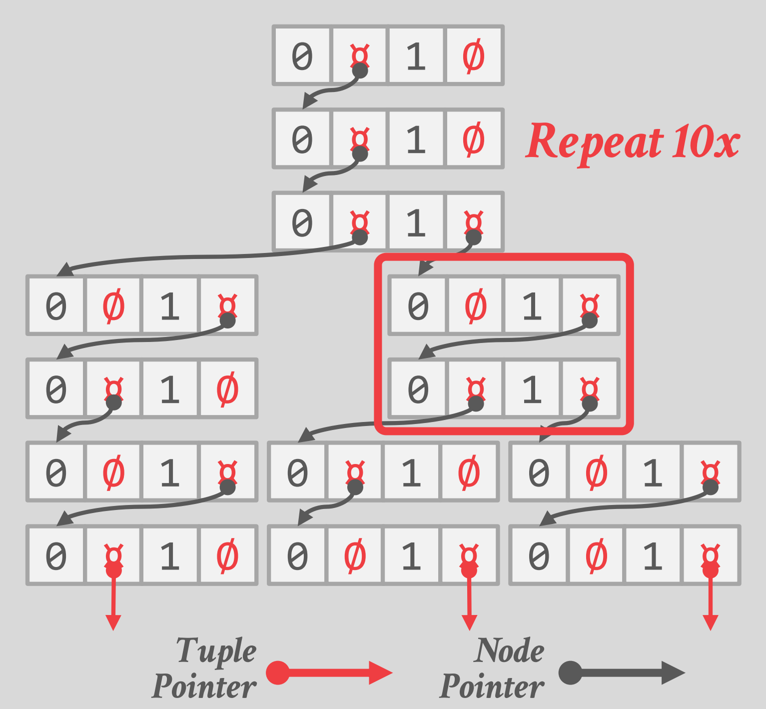
K10→ 00000000 0000$\boxed{1010}$

K25→ 00000000 00011001

K31→ 00000000 00011111

TRIE KEY SPAN

1-bit Span Trie



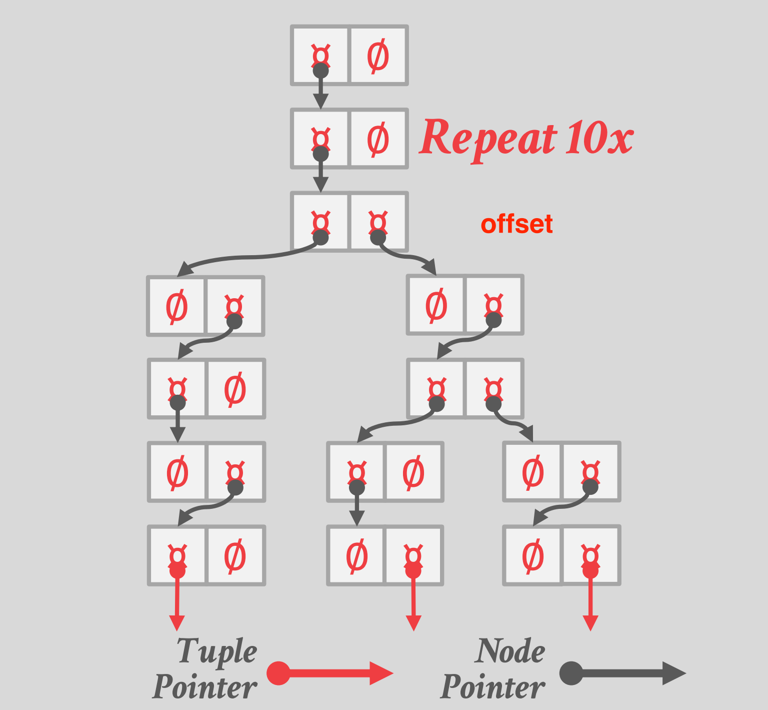
K10→ 00000000 00001010

K25→ 00000000 00011001

K31→ 00000000 00011111

TRIE KEY SPAN

1-bit Span Trie



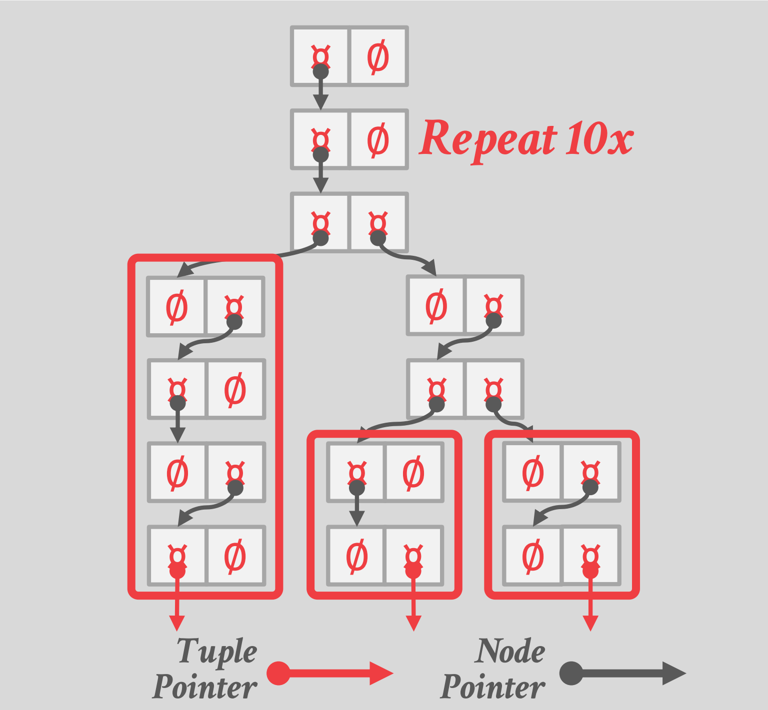
K10→ 00000000 00001010

K25→ 00000000 00011001

K31→ 00000000 00011111

TRIE KEY SPAN

1-bit Span Trie



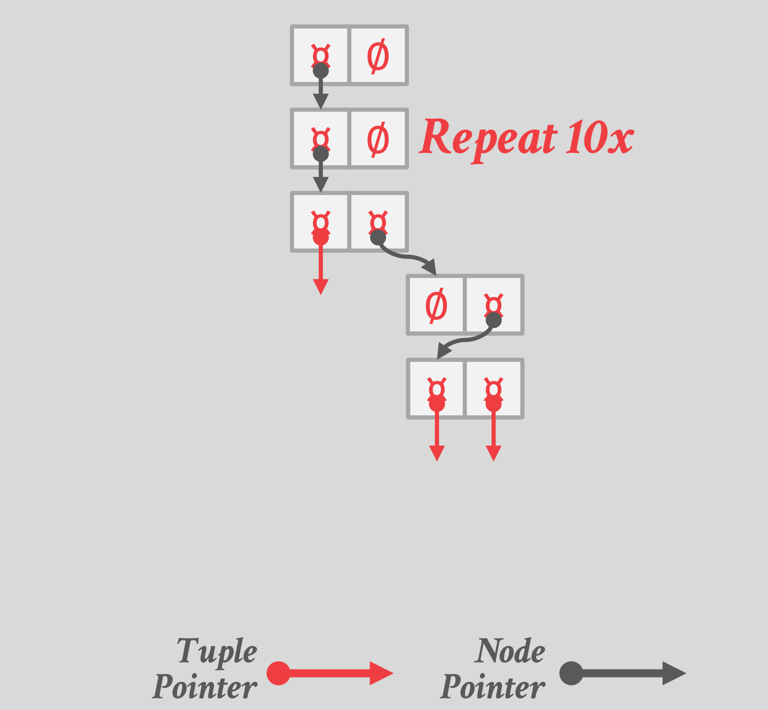
K10→ 00000000 00001010

K25→ 00000000 00011001

K31→ 00000000 00011111

RADIX TREE

1-bit Span Radix Tree

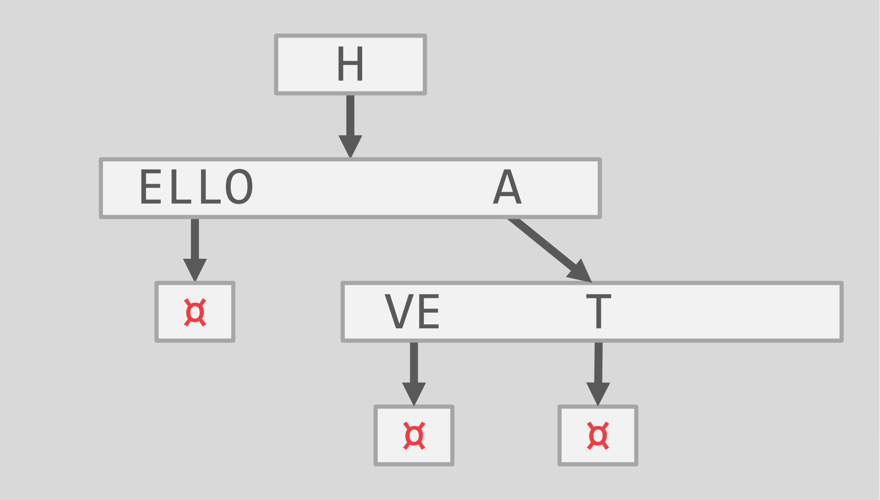


Omit all nodes with only a single child.

→ Also known as ***Patricia Tree***.

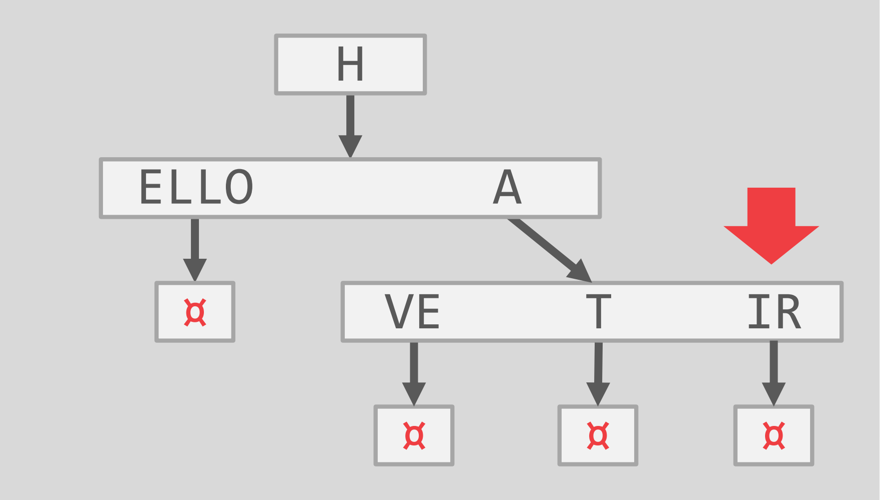
Can produce false positives, so the DBMS always checks the original tuple to see whether a key matches.

RADIX TREE: MODIFICATIONS



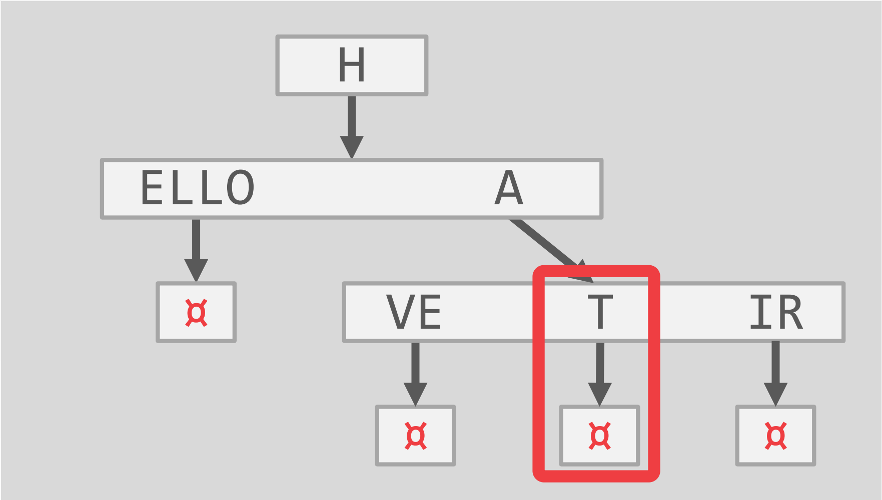
Insert HAIR

RADIX TREE: MODIFICATIONS



Insert HAIR

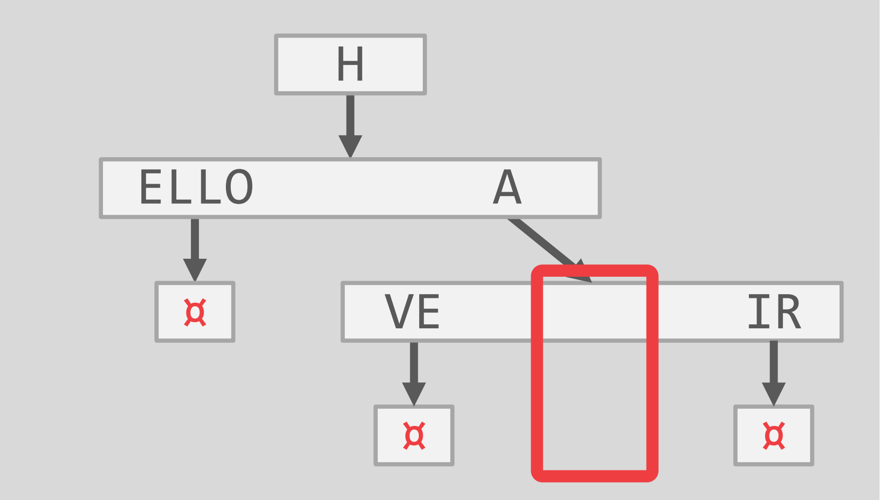
RADIX TREE: MODIFICATIONS



Insert HAIR

Delete HAT

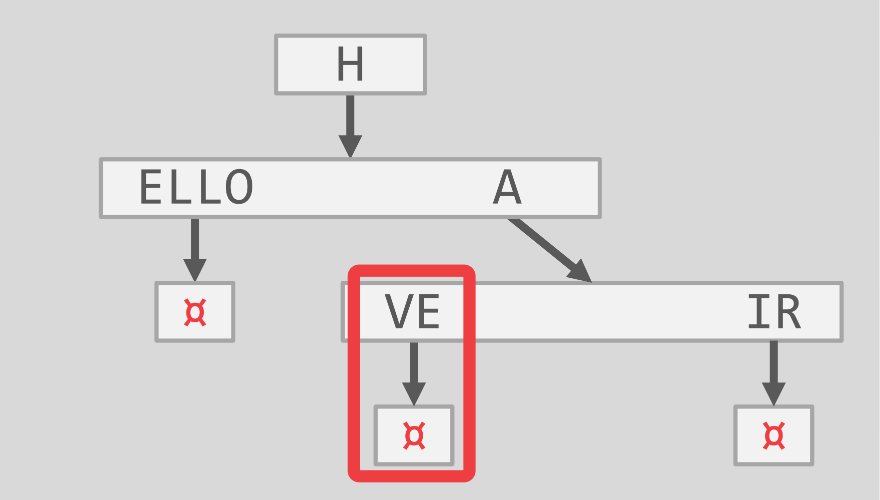
RADIX TREE: MODIFICATIONS



Insert HAIR

Delete HAT

RADIX TREE: MODIFICATIONS

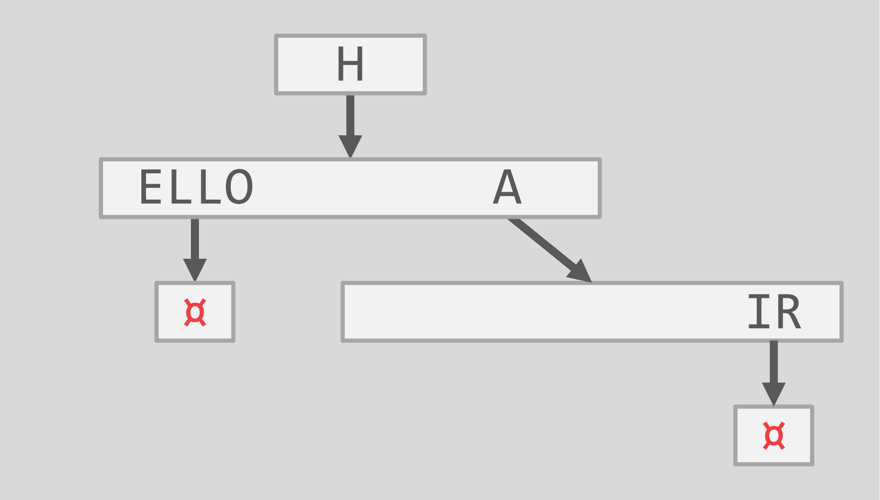


Insert HAIR

Delete HAT

Delete HAVE

RADIX TREE: MODIFICATIONS

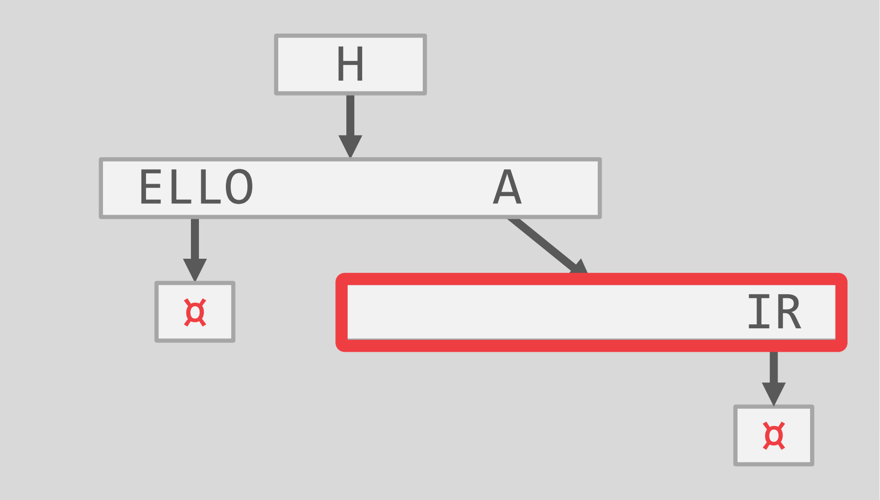


Insert HAIR

Delete HAT

Delete HAVE

RADIX TREE: MODIFICATIONS

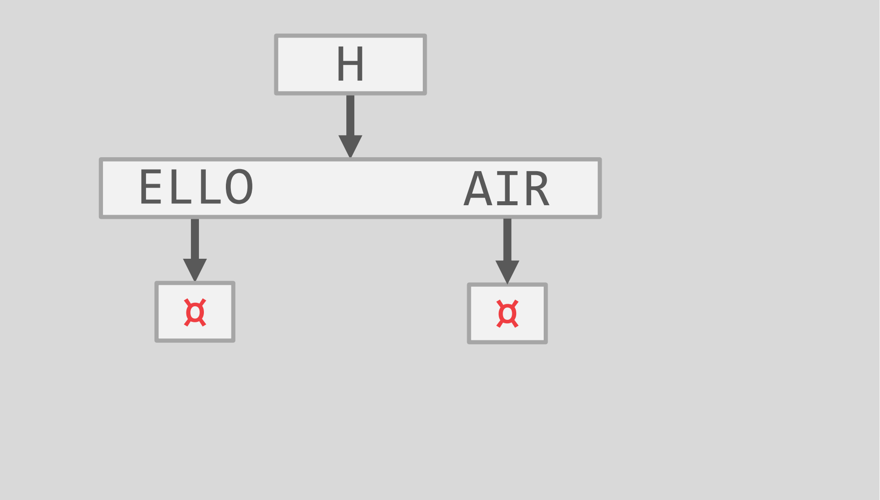


Insert HAIR

Delete HAT

Delete HAVE

RADIX TREE: MODIFICATIONS



Insert HAIR

Delete HAT

Delete HAVE

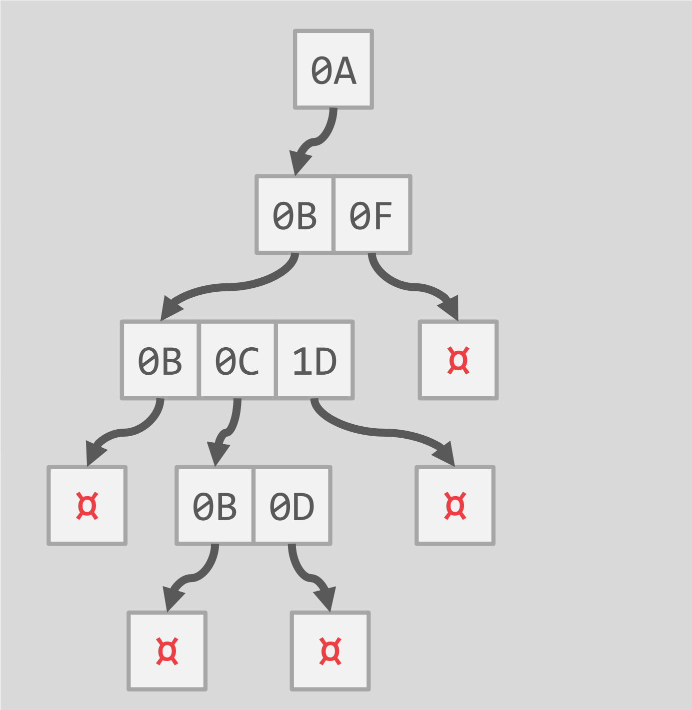
RADIX TREE: BINARY COMPARABLE KEYS

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.

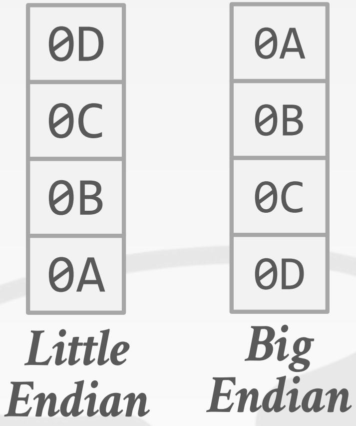
RADIX TREE: BINARY COMPARABLE KEYS

8-bit Span Radix Tree



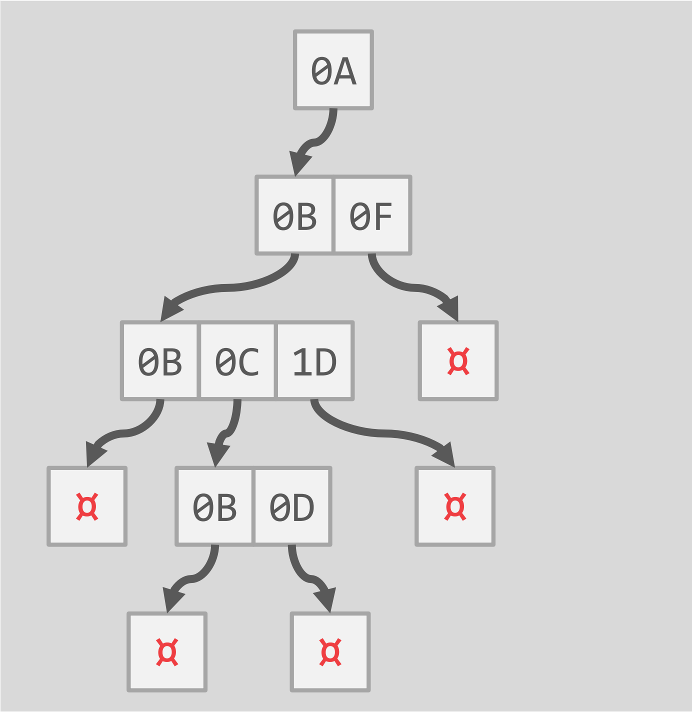
Int Key: 168496141

Hex Key: 0A 0B 0C 0D



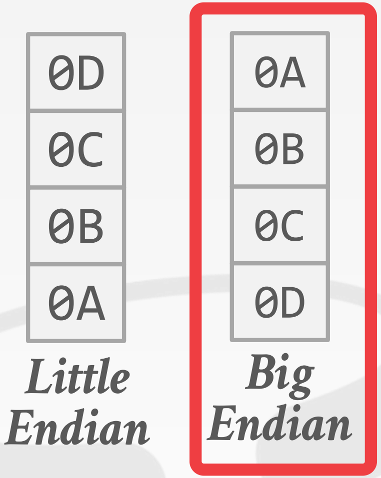
RADIX TREE: BINARY COMPARABLE KEYS

8-bit Span Radix Tree



Int Key: 168496141

Hex Key: 0A 0B 0C 0D

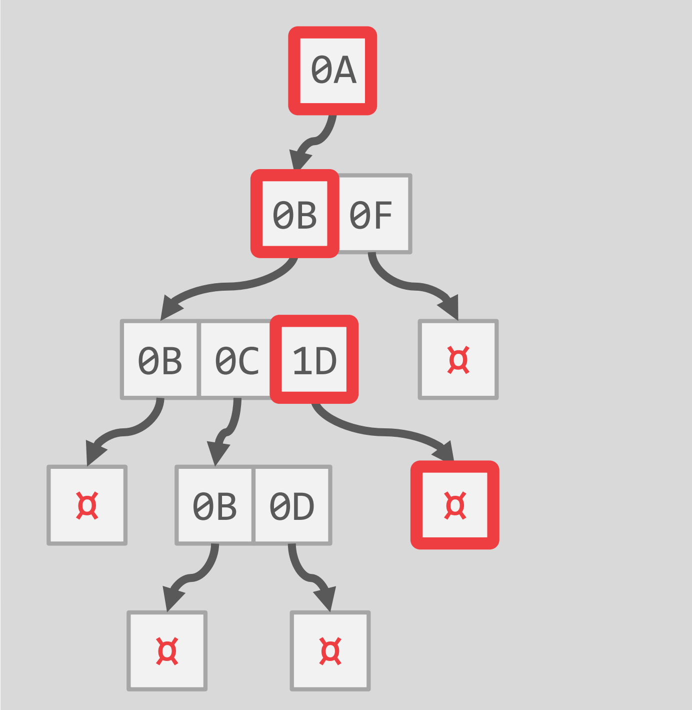


Find 658205

Hex 0A 0B 1D

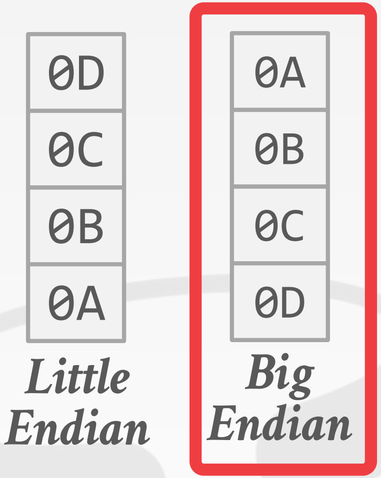
RADIX TREE: BINARY COMPARABLE KEYS

8-bit Span Radix Tree



Int Key: 168496141

Hex Key: 0A 0B 0C 0D



Find 658205

Hex 0A 0B 1D

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 **<u>not</u>** good at keyword searches:

→ Find all Wikipedia articles that contain the word "Pavlo"

Find a sub-element of above value for an attribute

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!