Carnegie Mellon University

Tree Indexes (Part I)

Lecture #06

Database Systems
15-445/15-645
Fall 2018

AP
Andy Pavlo
Computer Science
Carnegie Mellon Univ.

DATA STRUCTURES

Internal Meta-data

Core Data Storage

Temporary Data Structures

Table Indexes

TABLE INDEXES

A **table index** is a replica of a subset of a table's columns that are organized and/or sorted for efficient access using a subset of those columns.

The DBMS ensures that the contents of the table and the index are logically in sync.

TABLE INDEXES

It is the DBMS's job to figure out the best index(es) to use to execute each query.

There is a trade-off on the number of indexes to create per database.

→ Storage Overhead
→ Maintenance Overhead

TODAY'S AGENDA

B+Tree Overview

Design Decisions

Optimizations

B-TREE FAMILY

There is a specific data structure called a **B-Tree**, but then people also use the term to generally refer to a class of data structures.

→ **B-Tree**
→ **B+Tree**
→ **B^{link}-Tree**
→ **B*Tree**

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 systems that read and write large blocks of data.

The Ubiquitous B-Tree

DOUGLAS COMER

Computer Science Department, Purdue University, West Lafayette, Indiana 47907

B-trees have become, de facto, a standard for file organization. File indexes of users, dedicated database systems, and general-purpose access methods have all been proposed and implemented using B-trees This paper reviews B-trees and shows why they have been so successful It discusses the major variations of the B-tree, especially the B^+-tree, contrasting the relative merits and costs of each implementation. It illustrates a general purpose access method which uses a B-tree.

Keywords and Phrases: B-tree, B^*-tree, B^+-tree, file organization, index

CR Categories: 3.73 3.74 4.33 4 34

INTRODUCTION

The secondary storage facilities available on large computer systems allow users to store, update, and recall data from large collections of information called files. A computer must retrieve an item and place it in main memory before it can be processed. In order to make good use of the computer resources, one must organize files intelligently, making the retrieval process efficient.

The choice of a good file organization depends on the kinds of retrieval to be performed. There are two broad classes of retrieval commands which can be illustrated by the following examples:

Sequential: "From our employee file, prepare a list of all employees' names and addresses," and

Random: "From our employee file, extract the information about employee J. Smith".

We can imagine a filing cabinet with three drawers of folders, one folder for each employee. The drawers might be labeled "A–G," "H–R," and "S–Z," while the folders might be labeled with the employees' last names. A sequential request requires the searcher to examine the entire file, one folder at a time. On the other hand, a random request implies that the searcher, guided by the labels on the drawers and folders, need only extract one folder.

Associated with a large, randomly accessed file in a computer system is an *index* which, like the labels on the drawers and folders of the file cabinet, speeds retrieval by directing the searcher to the small part of the file containing the desired item. Figure 1 depicts a file and its index. An index may be physically integrated with the file, like the labels on employee folders, or physically separate, like the labels on the drawers. Usually the index itself is a file. If the index file is large, another index may be built on top of it to speed retrieval further, and so on. The resulting hierarchy is similar to the employee file, where the topmost index consists of labels on drawers, and the next level of index consists of labels on folders.

Natural hierarchies, like the one formed by considering last names as index entries, do not always produce the best perform-

Computing Surveys, Vol 11, No 2, June 1979

B+TREE PROPERTIES

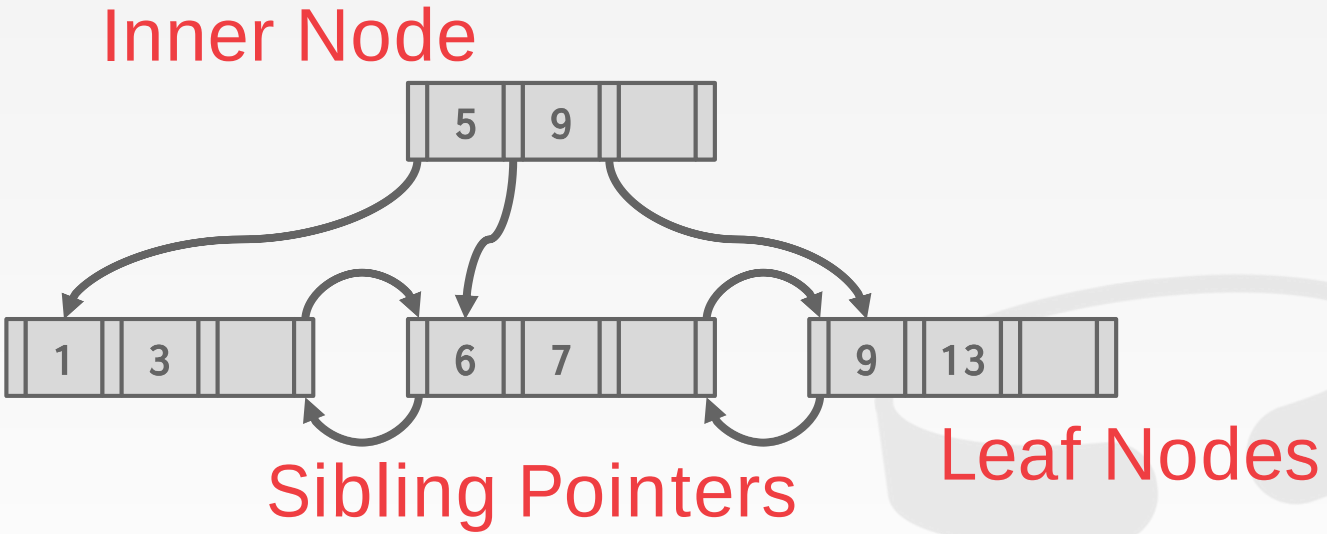
A B+tree is an ***M***-way search tree with the following properties:

→ It is perfectly balanced (i.e., every leaf node is at the same depth).

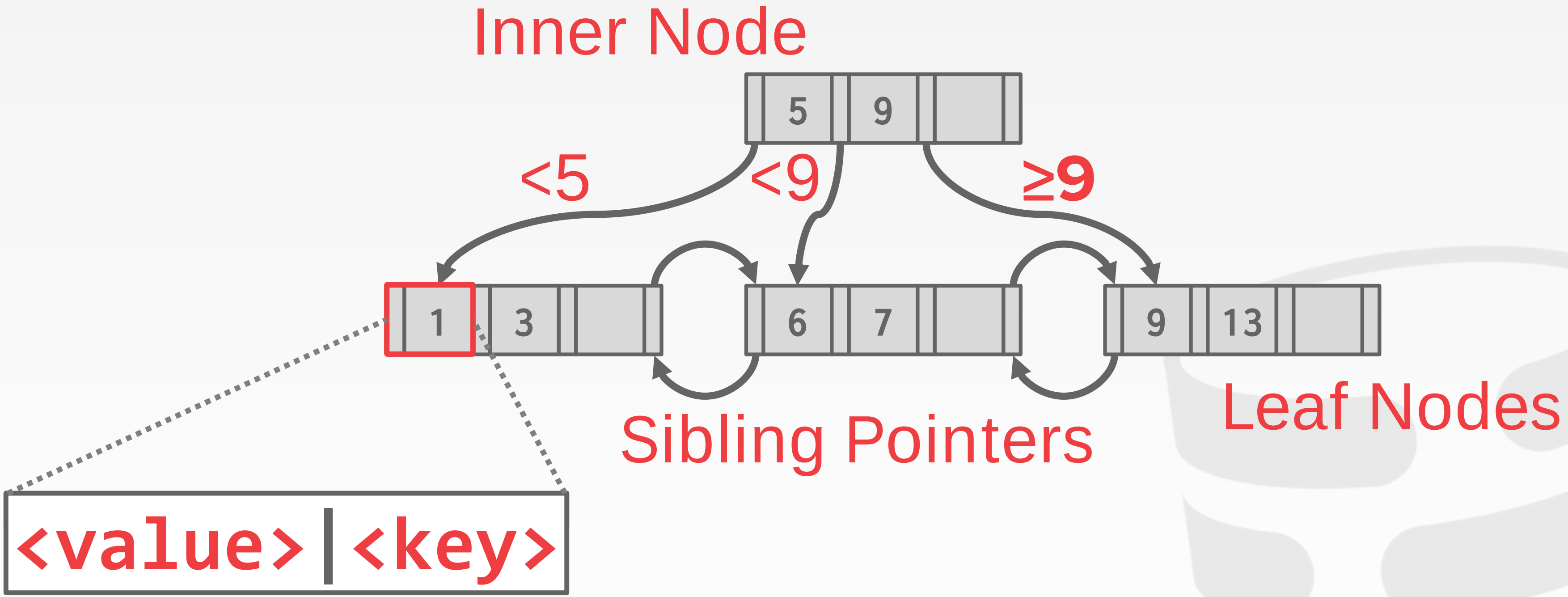
→ Every inner node other than the root, is at least half-full
`M/2-1 ≤ #keys ≤ M-1`

→ Every inner node with **`k`** keys has **`k+1`** non-null children

B+TREE EXAMPLE



B+TREE EXAMPLE



NODES

Every node in the B+Tree contains an array of key/value pairs.

→ The keys will always be the column or columns that you built your index on

→ The values will differ based on whether the node is classified as **inner nodes** or **leaf nodes.**

The arrays are (usually) kept in sorted key order.

LEAF NODE VALUES

Approach #1: Record Ids

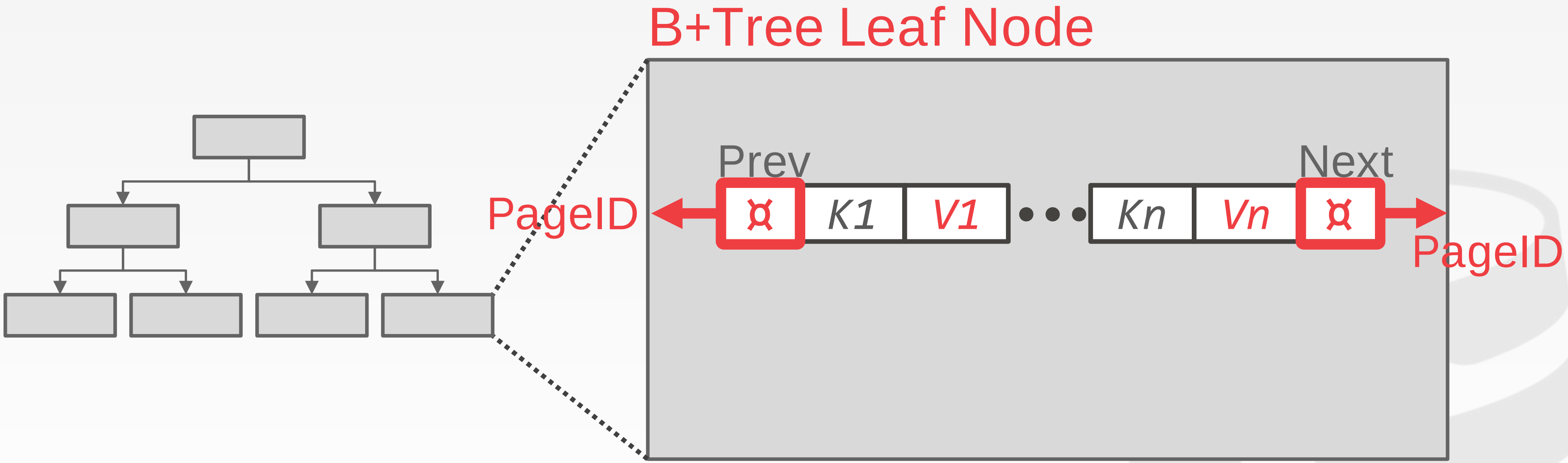
→ A pointer to the location of the tuple that the index entry corresponds to.

Approach #2: Tuple Data

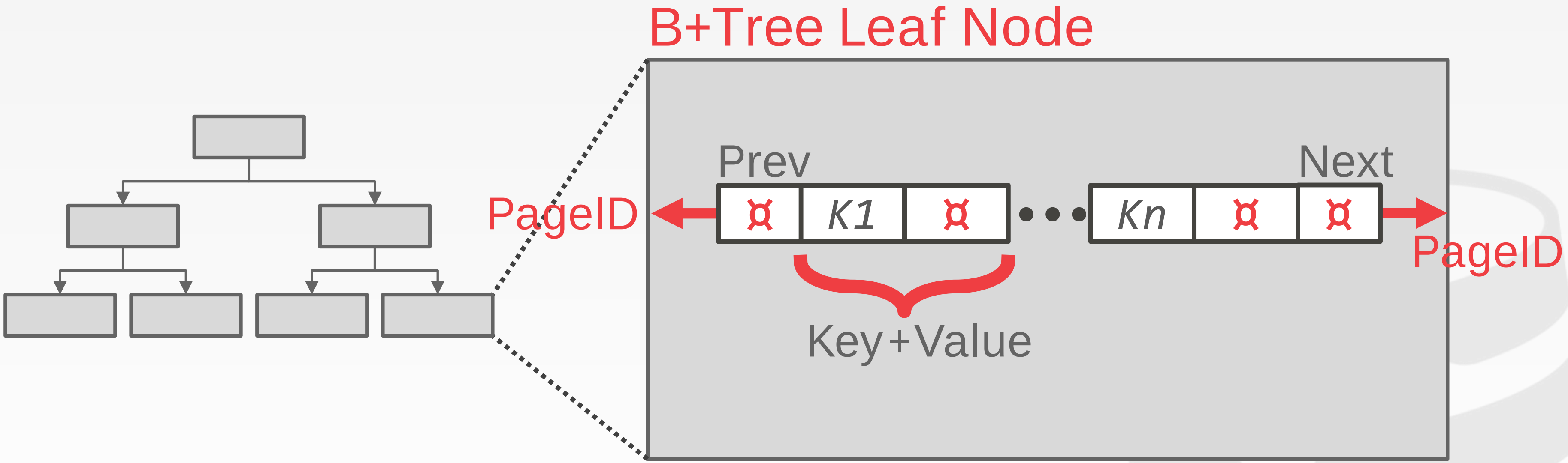
→ The actual contents of the tuple is stored in the leaf node.

→ Secondary indexes have to store the record id as their values.

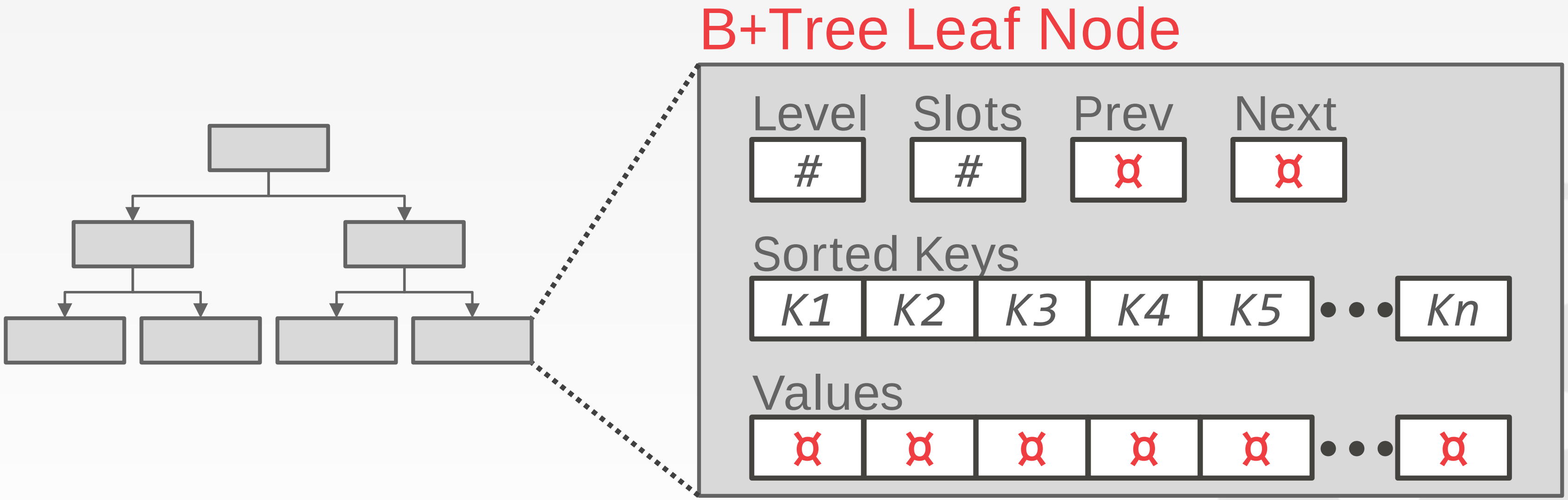
B+TREE LEAF NODES



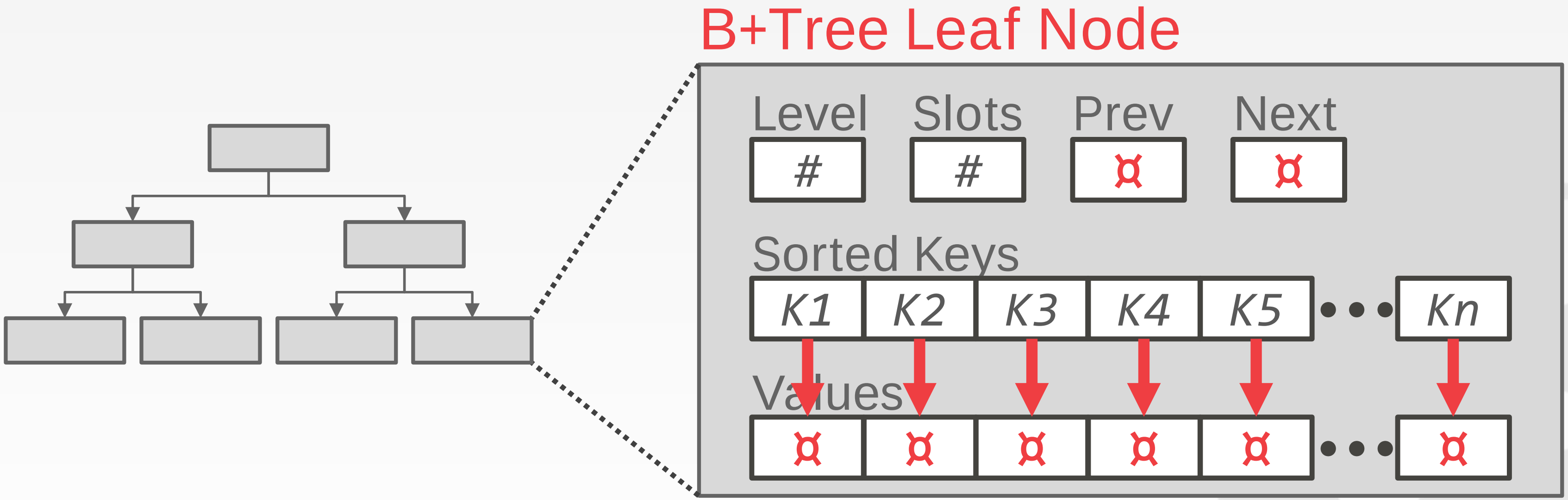
B+TREE LEAF NODES



B+TREE LEAF NODES



B+TREE LEAF NODES



B-TREE VS. B+TREE

The original **B-Tree** from 1972 stored keys + values in all nodes in the tree.

→ More space efficient since each key only appears once in the tree.

A **B+Tree** only stores values in leaf nodes. Inner nodes only guide the search process.

B+TREE INSERT

Find correct leaf **L**.
Put data entry into **L** in sorted order.

If **L** has enough space, done!

Else, must split **L** into **L** and a new node **L2**
→ Redistribute entries evenly, copy up middle key.
→ Insert index entry pointing to **L2** into parent of **L**.

To split inner node, redistribute entries evenly, but push up middle key.

Source: Chris Re

B+TREE VISUALIZATION

https://cmudb.io/btree

Source: David Gales (Univ. of San Francisco)

B+TREE DELETE

Start at root, find leaf **L** where entry belongs.
Remove the entry.
If **L** is at least half-full, done!
If **L** has only **M/2-1** entries,
- → Try to re-distribute, borrowing from sibling (adjacent node with same parent as **L**).
- → If re-distribution fails, merge **L** and sibling.

If merge occurred, must delete entry (pointing to **L** or sibling) from parent of **L**.

Source: Chris Re

B+TREES IN PRACTICE

Typical Fill-Factor: 67%.

→ Average Fanout = 2*100*0.67 = 134

Typical Capacities:

→ Height 4: 1334 = 312,900,721 entries
→ Height 3: 1333 = 2,406,104 entries

Pages per level:

→ Level 1 = 1 page = 8 KB
→ Level 2 = 134 pages = 1 MB
→ Level 3 = 17,956 pages = 140 MB

CLUSTERED INDEXES

The table is stored in the sort order specified by the primary key.

→ Can be either heap- or index-organized storage.

Some DBMSs always use a clustered index.

→ If a table doesn't include a pkey, the DBMS will automatically make a hidden row id pkey.

Other DBMSs cannot use them at all.

SELECTION CONDITIONS

The DBMS can use a B+Tree index if the query provides any of the attributes of the search key.

Example: Index on **<a,b,c>**

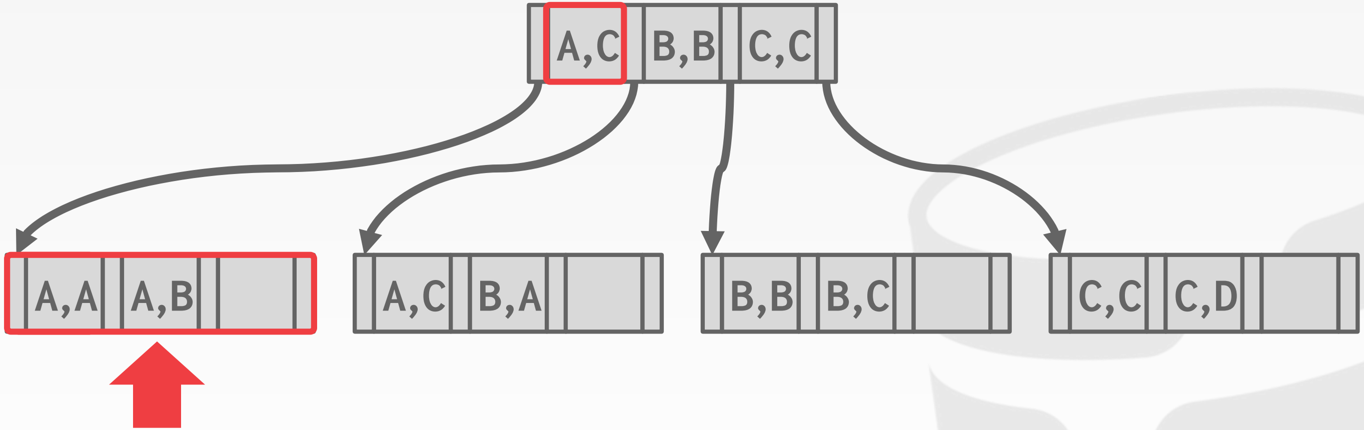
→ Supported: **(a=5 AND b=3)**
→ Supported: **(b=3)**.

Not all DBMSs support this.

For hash index, we must have all attributes in search key.

SELECTION CONDITIONS

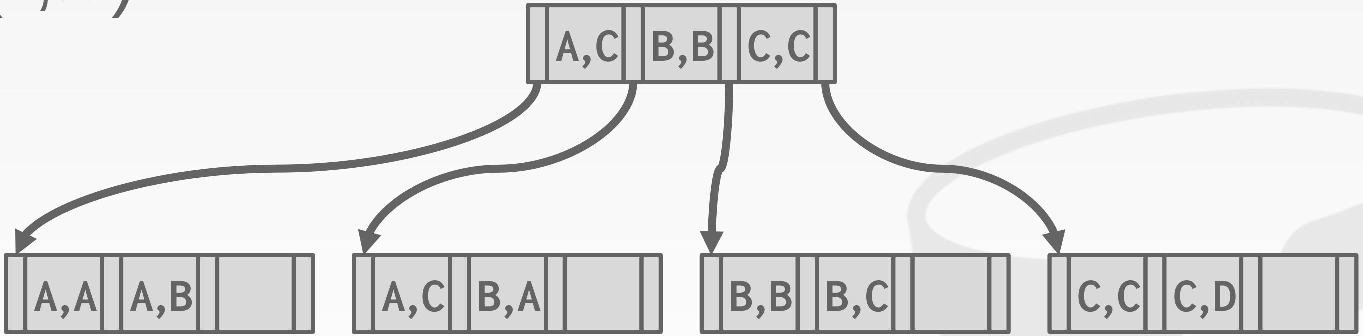
Find Key=(A,B)



SELECTION CONDITIONS

Find Key=(A,B)

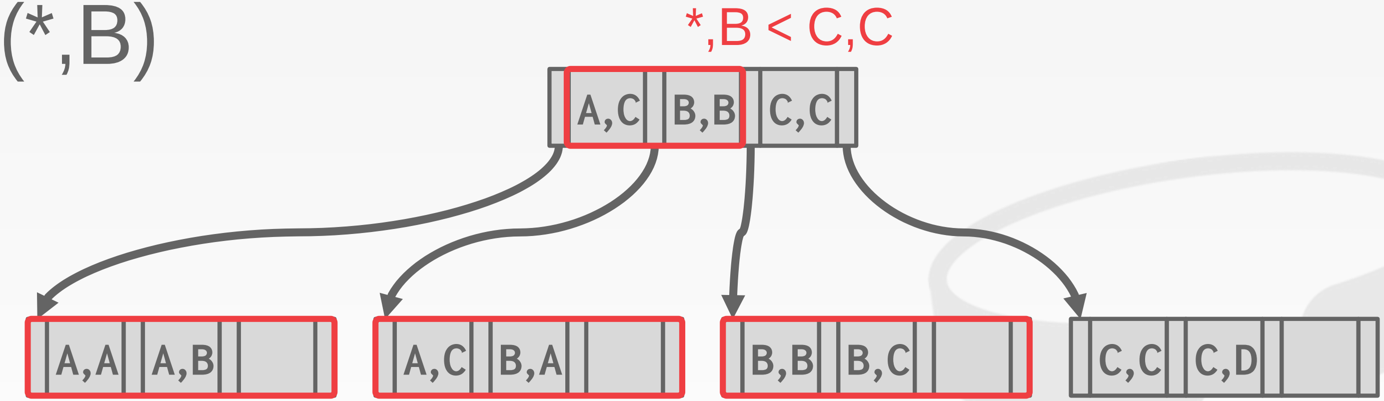
Find Key=(*,B)



SELECTION CONDITIONS

Find Key=(A,B)

Find Key=(*,B)



B+TREE DESIGN CHOICES

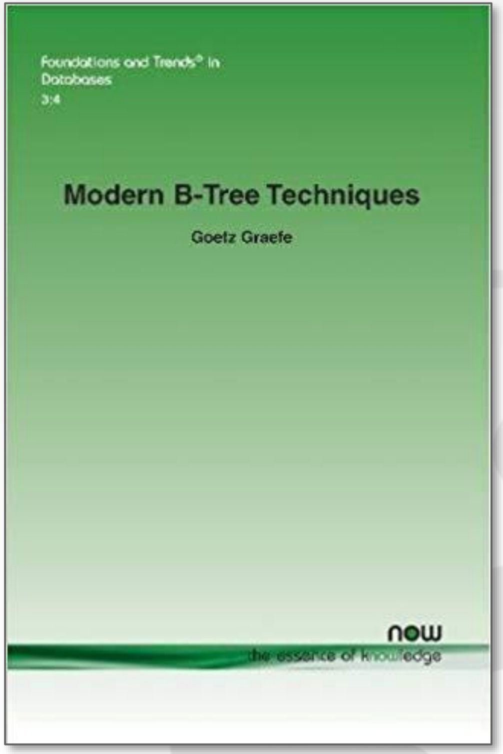
Node Size

Merge Threshold

Variable Length Keys

Non-Unique Indexes

Intra-Node Search



NODE SIZE

The slower the disk, 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 don't always merge nodes when it is half full.

Delaying a merge operation may reduce the amount of reorganization.

May 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 B+Tree can vary.

→ Requires careful memory management.

Approach #3: Key Map

→ Embed an array of pointers that map to the key + value list within the node.

NON-UNIQUE INDEXES

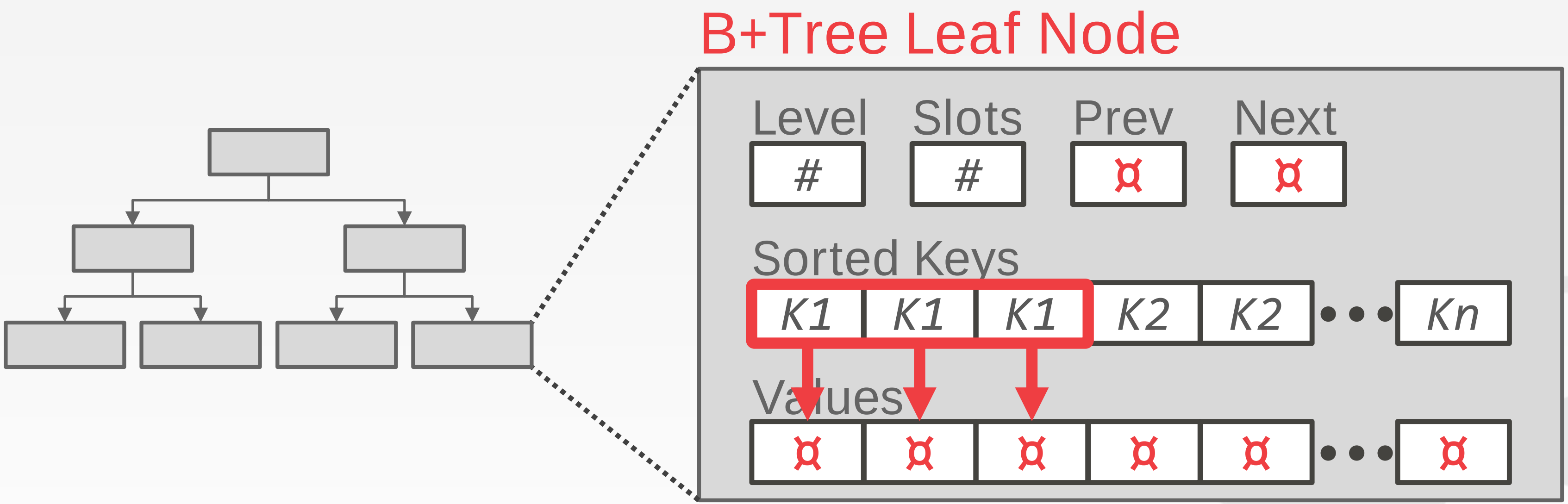
Approach #1: Duplicate Keys

→ Use the same leaf node layout but store duplicate keys multiple times.

Approach #2: Value Lists

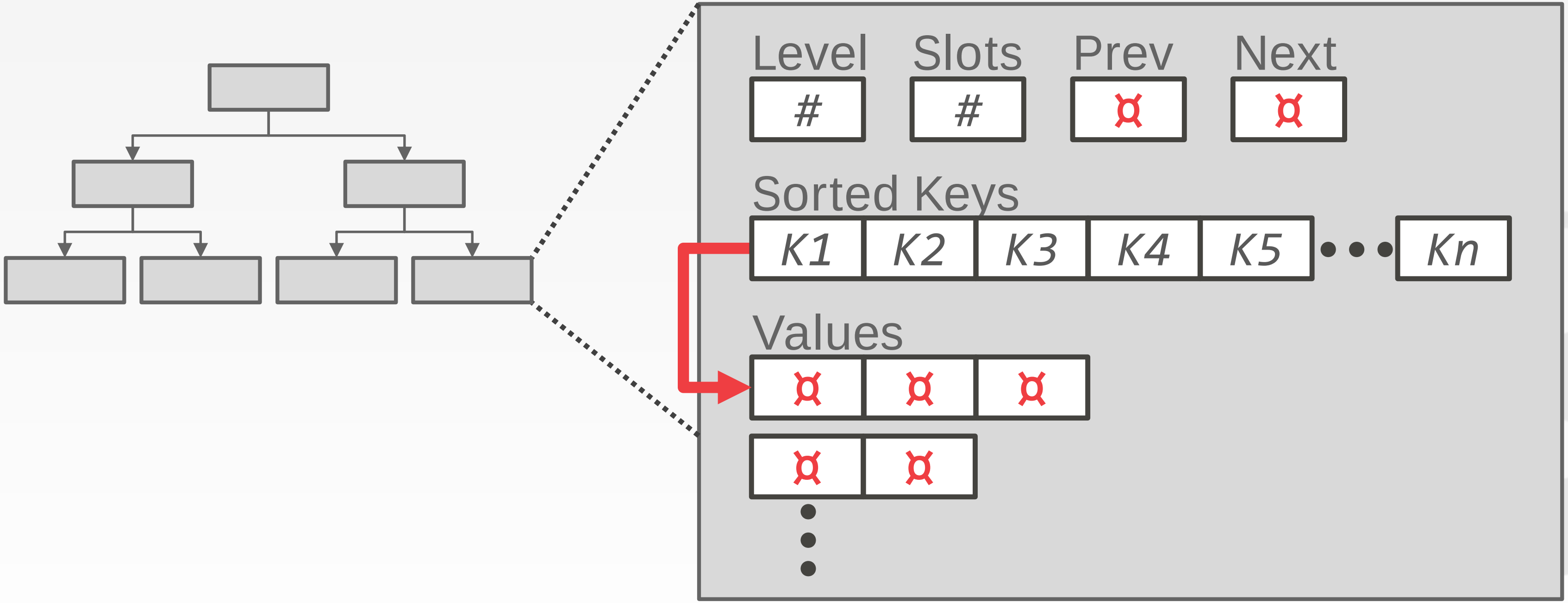
→ Store each key only once and maintain a linked list of unique values.

NON-UNIQUE: DUPLICATE KEYS



NON-UNIQUE: VALUE LISTS

B+Tree Leaf Node



INTRA-NODE SEARCH

Find Key=8

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.

INTRA-NODE SEARCH

Approach #1: Linear

→ Scan node keys from beginning to end.

Approach #2: Binary

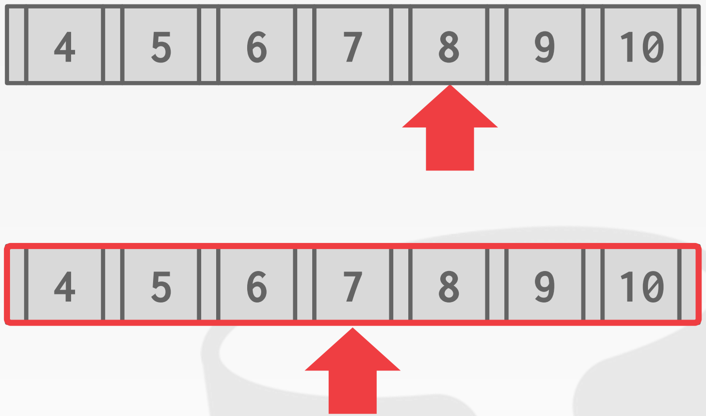
→ Jump to middle key, pivot left/right depending on comparison.

Approach #3: Interpolation

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

INTRA-NODE SEARCH

Find Key=8



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.

INTRA-NODE SEARCH

Approach #1: Linear

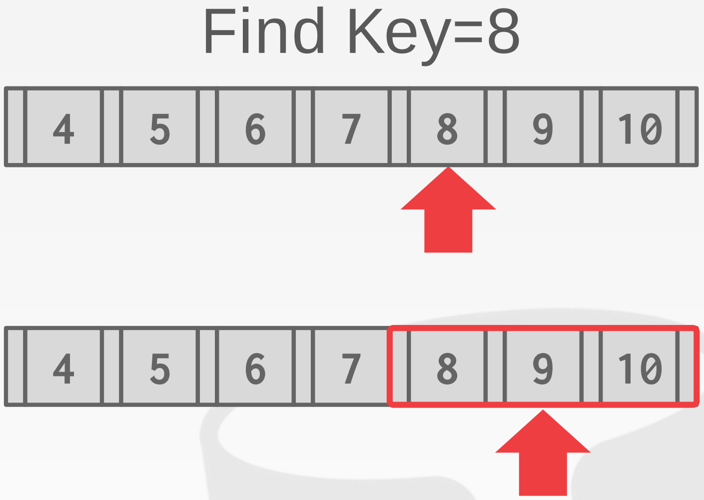
→ Scan node keys from beginning to end.

Approach #2: Binary

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

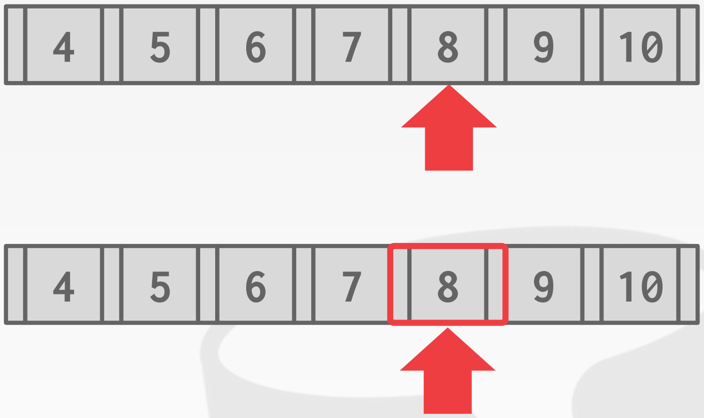
Approach #3: Interpolation

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



INTRA-NODE SEARCH

Find Key=8



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.

INTRA-NODE SEARCH

Approach #1: Linear

→ Scan node keys from beginning to end.

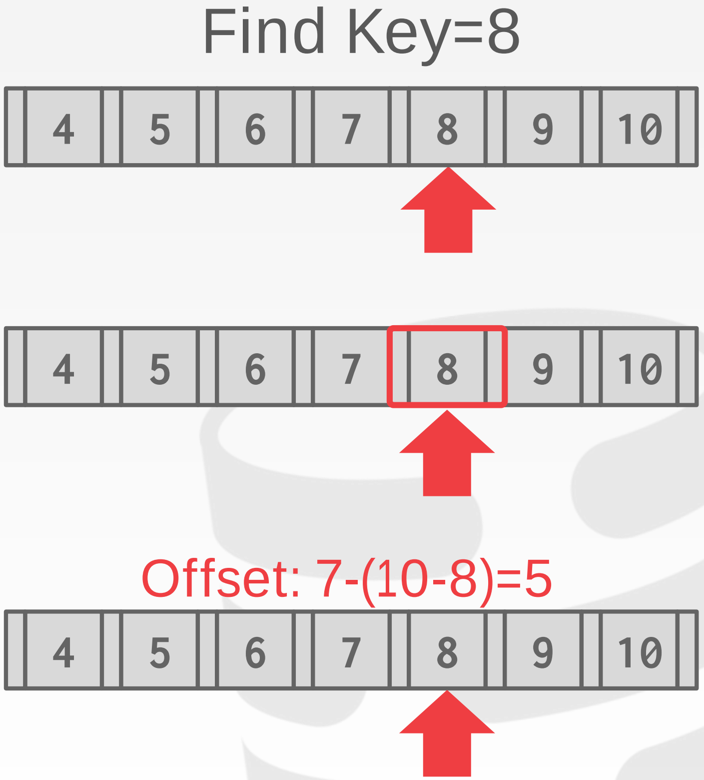
Approach #2: Binary

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

Approach #3: Interpolation

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

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

Suffix Truncation

Bulk Insert

Pointer Swizzling

PREFIX COMPRESSION

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

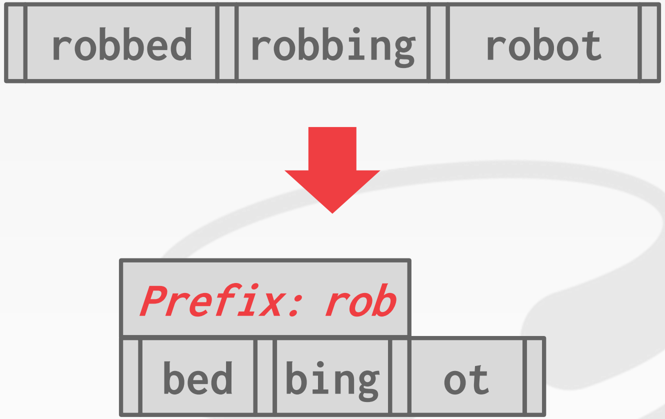
Instead of storing the entire key each time, extract common prefix and store only unique suffix for each key.

→ Many variations.

PREFIX COMPRESSION

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

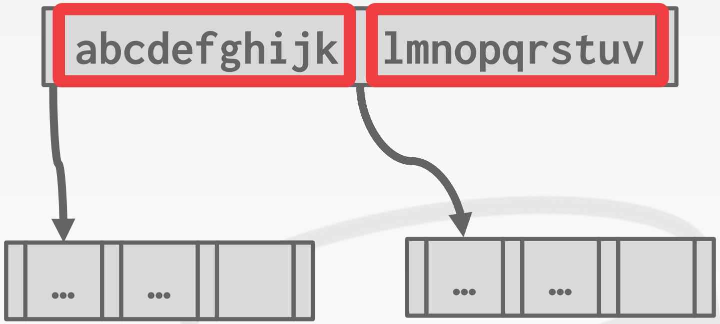
Instead of storing the entire key each time, extract common prefix and store only unique suffix for each key.
→ Many variations.



SUFFIX TRUNCATION

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

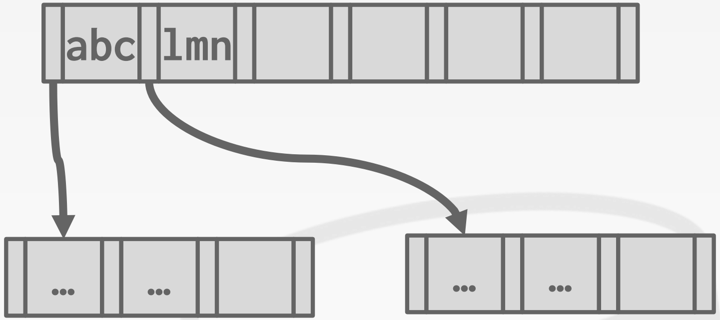
Store a minimum prefix that is needed to correctly route probes into the index.



SUFFIX TRUNCATION

The keys in the inner nodes are only used to "direct traffic".

→ We don't actually need the entire key.



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

BULK INSERT

The fastest/best way to build a B+Tree is to first sort the keys and then build the index from the bottom up.

BULK INSERT

The fastest/best way to build a B+Tree is to first sort the keys and then build the index from the bottom up.

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

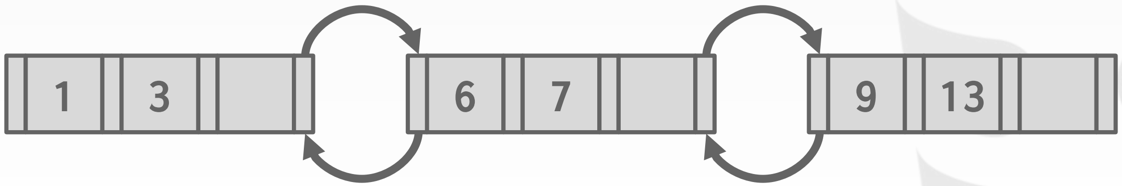
Sorted Keys: 1, 3, 6, 7, 9, 13

BULK INSERT

The fastest/best way to build a B+Tree 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

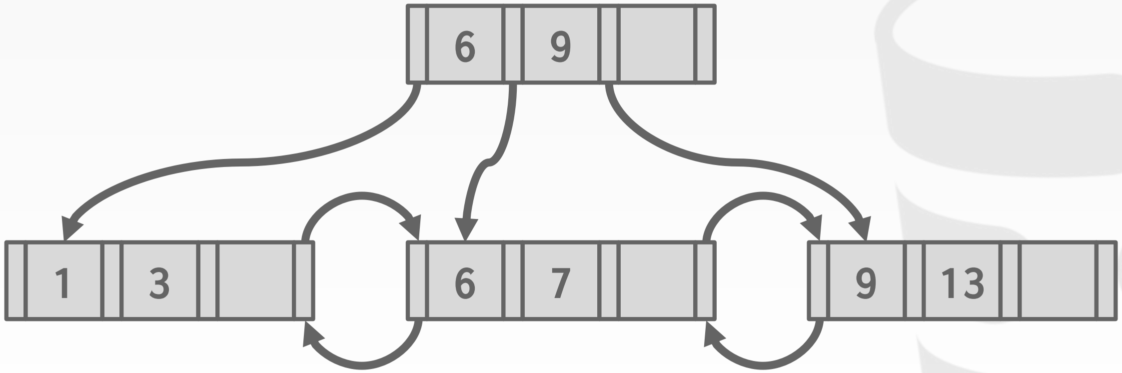


BULK INSERT

The fastest/best way to build a B+Tree 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

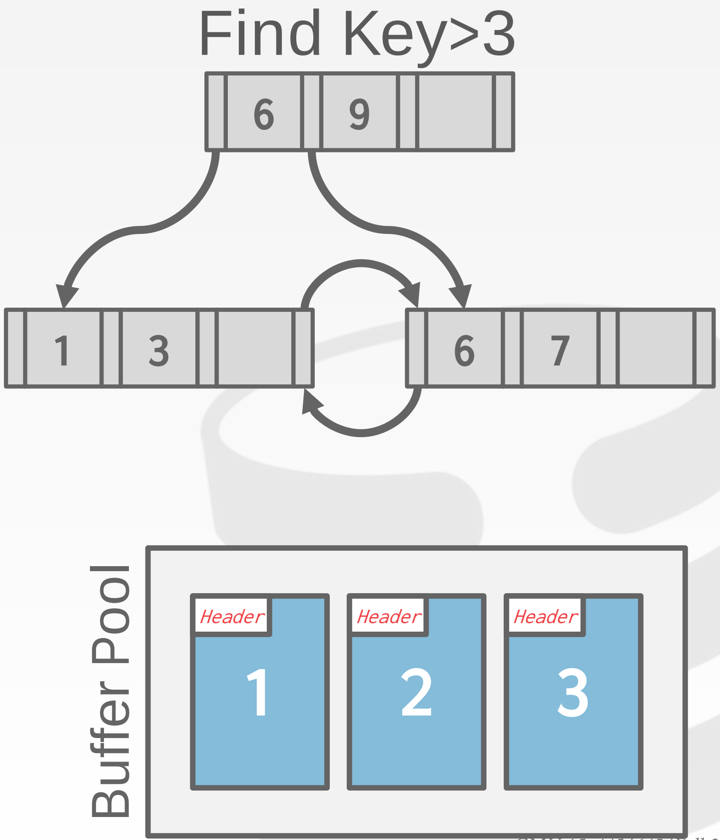
Nodes use page ids to reference other nodes in the index. The DBMS has to get the memory location from the page table during traversal.

If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids, thereby removing the need to get address from the page table.

POINTER SWIZZLING

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

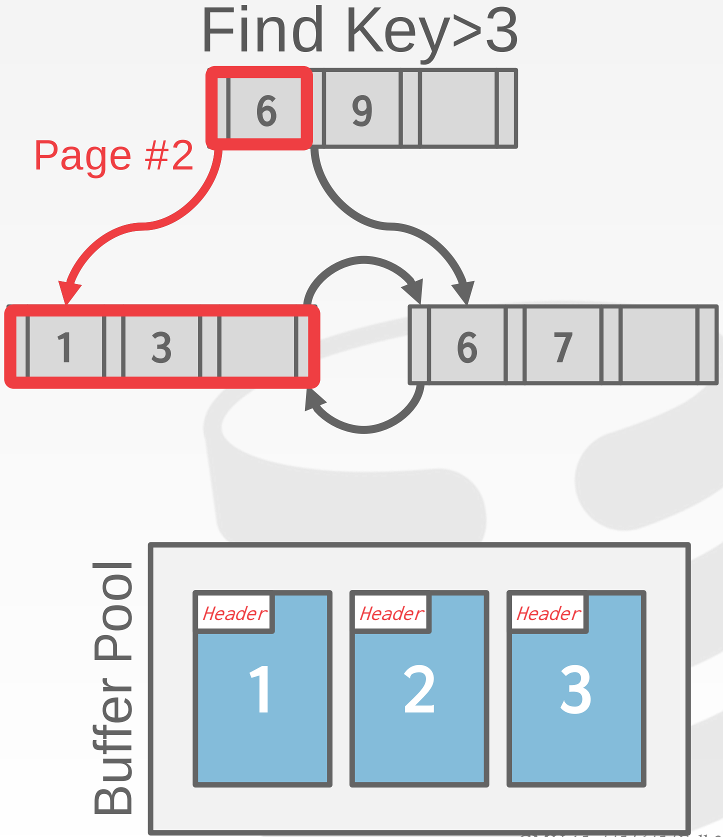
If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids, thereby removing the need to get address from the page table.



POINTER SWIZZLING

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

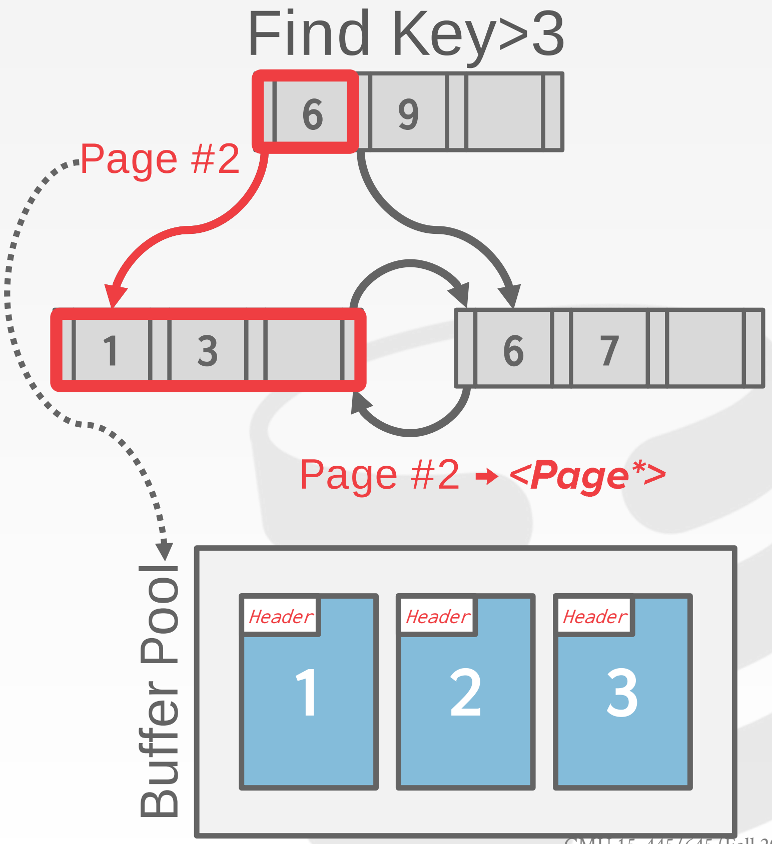
If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids, thereby removing the need to get address from the page table.



POINTER SWIZZLING

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

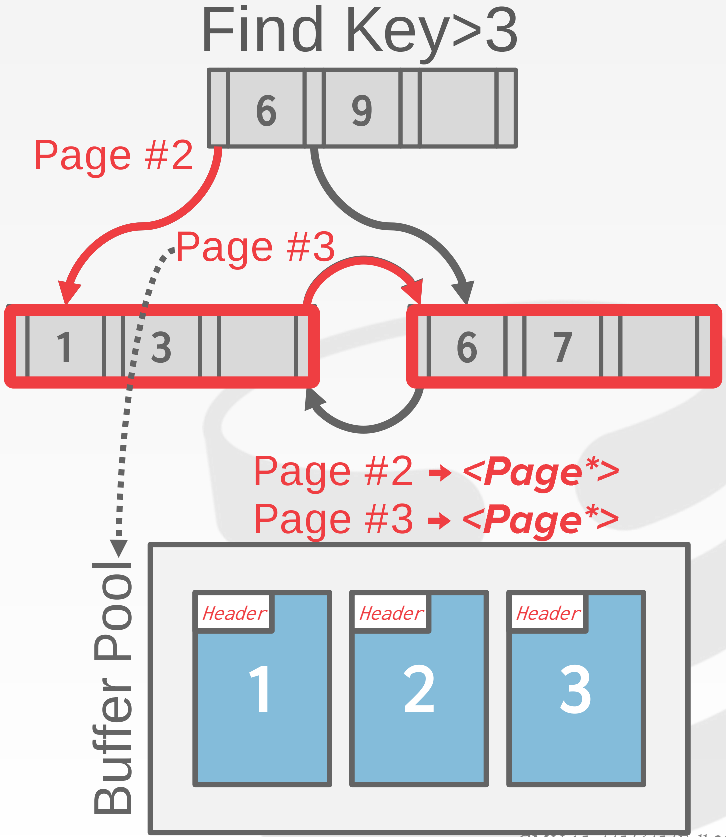
If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids, thereby removing the need to get address from the page table.



POINTER SWIZZLING

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

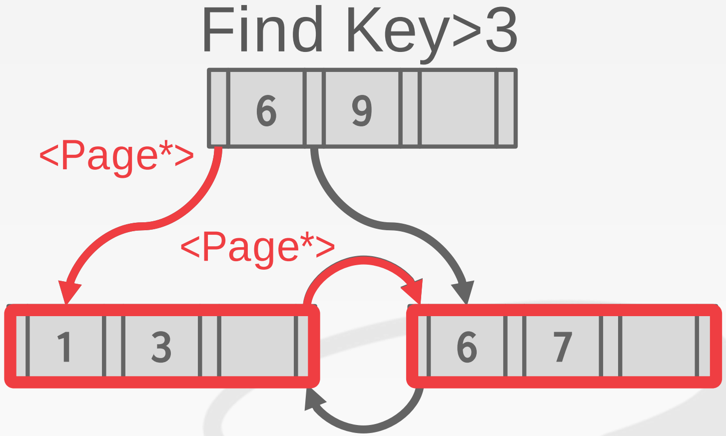
If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids, thereby removing the need to get address from the page table.

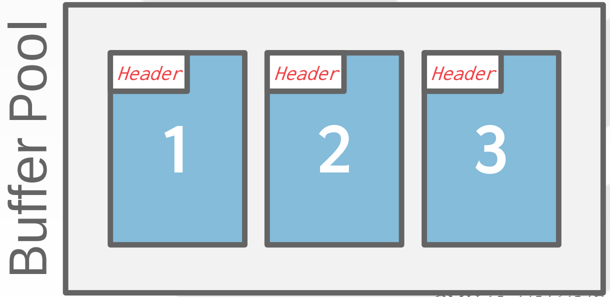


POINTER SWIZZLING

Nodes use page ids to reference other nodes in the index. The DBMS has to 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, thereby removing the need to get address from the page table.





CONCLUSION

The venerable B+Tree is always a good choice for your DBMS.

NEXT CLASS

Skip Lists

Radix Trees

Inverted Indexes