University of Michigan

# 16 Tree Indexes

Database Management Systems
EECS 484
Fall 2024

LM
Lin Ma
Computer Science and
Engineering Division

# DATA STRUCTURES

Internal Meta-data

Core Data Storage

Temporary Data Structures

Table Indexes

# DATA STRUCTURES

Internal Meta-data

Core Data Storage

Temporary Data Structures

$\boxed{\text{Table Indexes}}$

# TABLE INDEXES

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

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

# 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 regarding the number of indexes to create per database.
→ Storage Overhead
→ Maintenance Overhead

# TODAY'S AGENDA

B+Tree Overview

Use in a DBMS (Demo)

Design Choices

Optimizations

# B-TREE FAMILY

There is a specific data structure called a **B-Tree**.

People also use the term to generally refer to a class of balanced tree data structures:

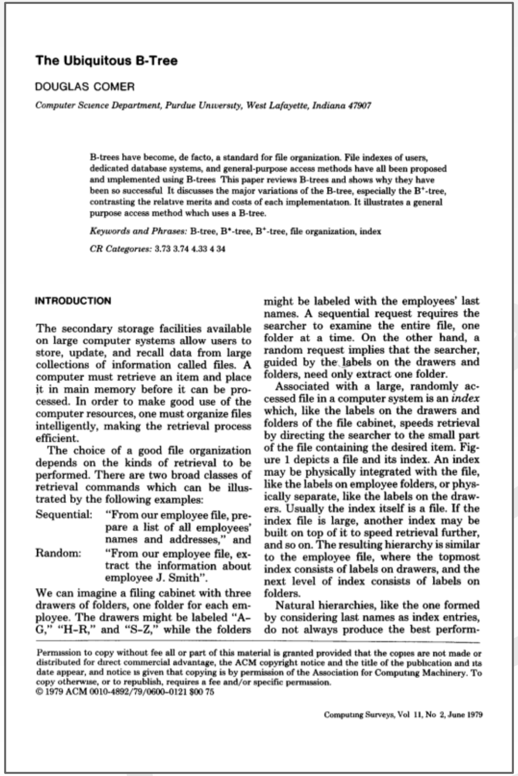
→ **B-Tree** (1971)
→ **B+Tree** (1973)
→ **B*Tree** (1977?)
→ **B$^{link}$-Tree** (1981)

# B+TREE

A **B+Tree** is a self-balancing tree data structure that keeps data sorted and allows searches, insertions, and deletions in **O(log n)**.

→ Generalization of a binary search tree, since 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-

# B+TREE PROPERTIES

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

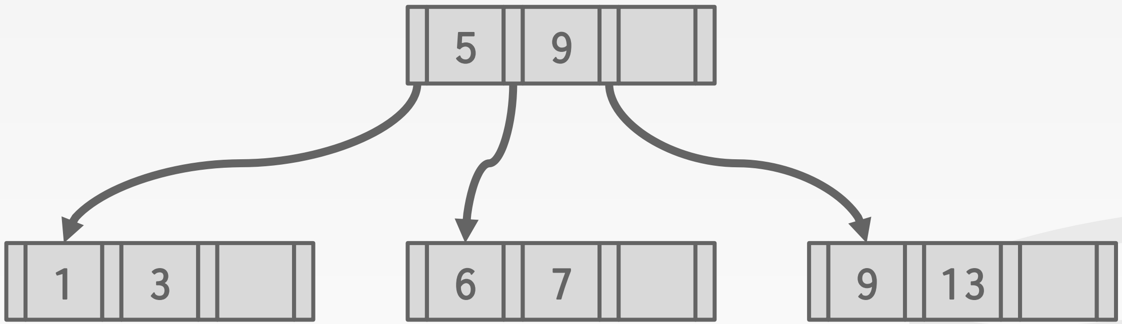
→ It is perfectly balanced (i.e., every leaf node is at the same depth in the tree)

→ Every node other than the root is at least half-full

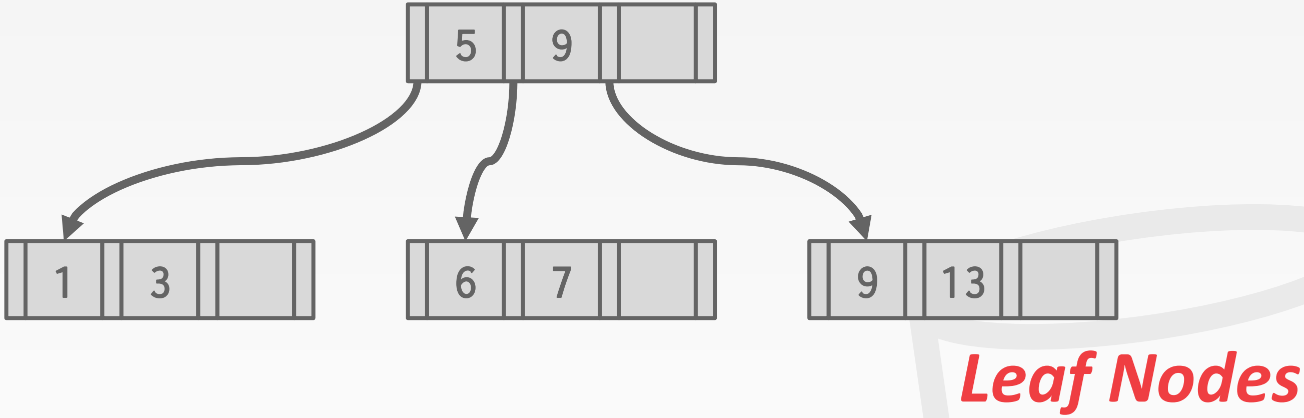
`M/2-1 ≤ #keys ≤ M-1`

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

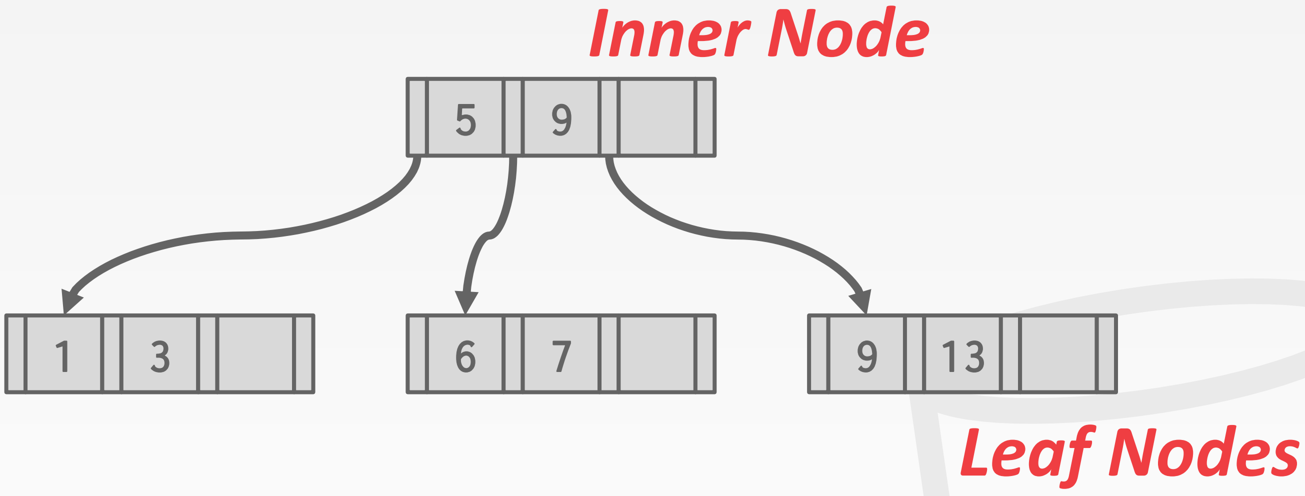
# B+TREE EXAMPLE



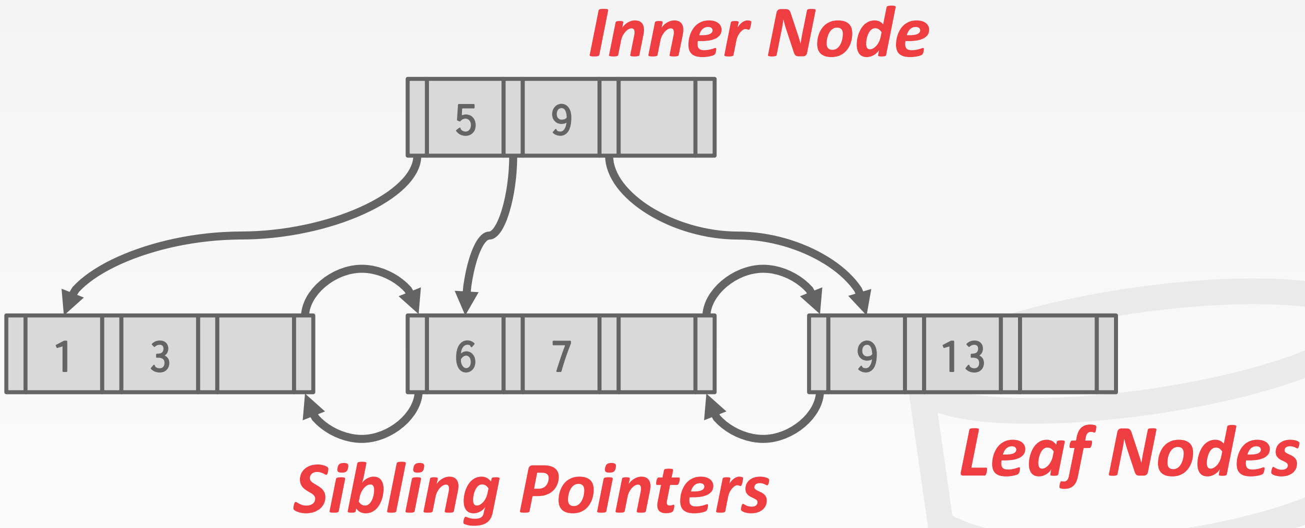
# B+TREE EXAMPLE



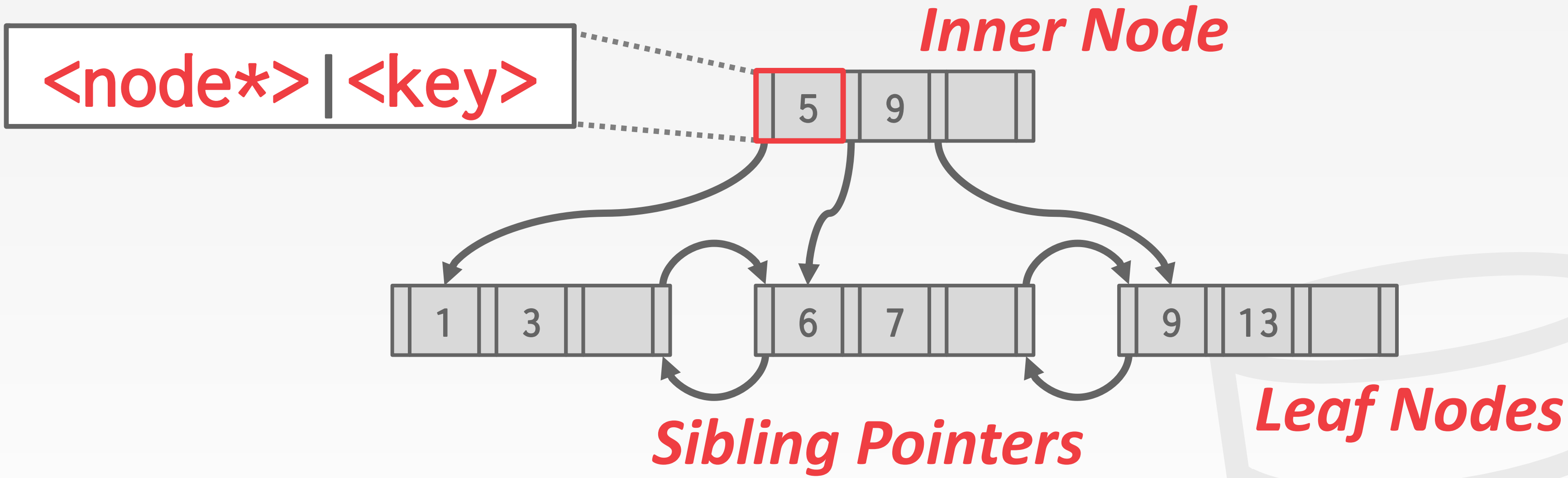
# B+TREE EXAMPLE



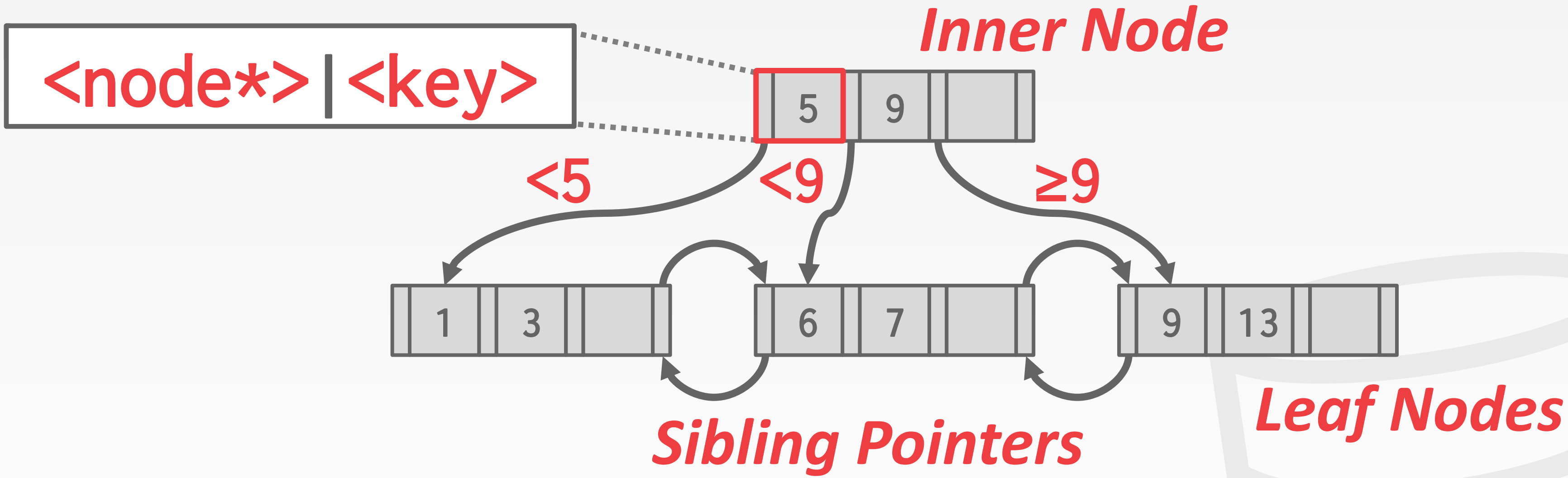
# B+TREE EXAMPLE



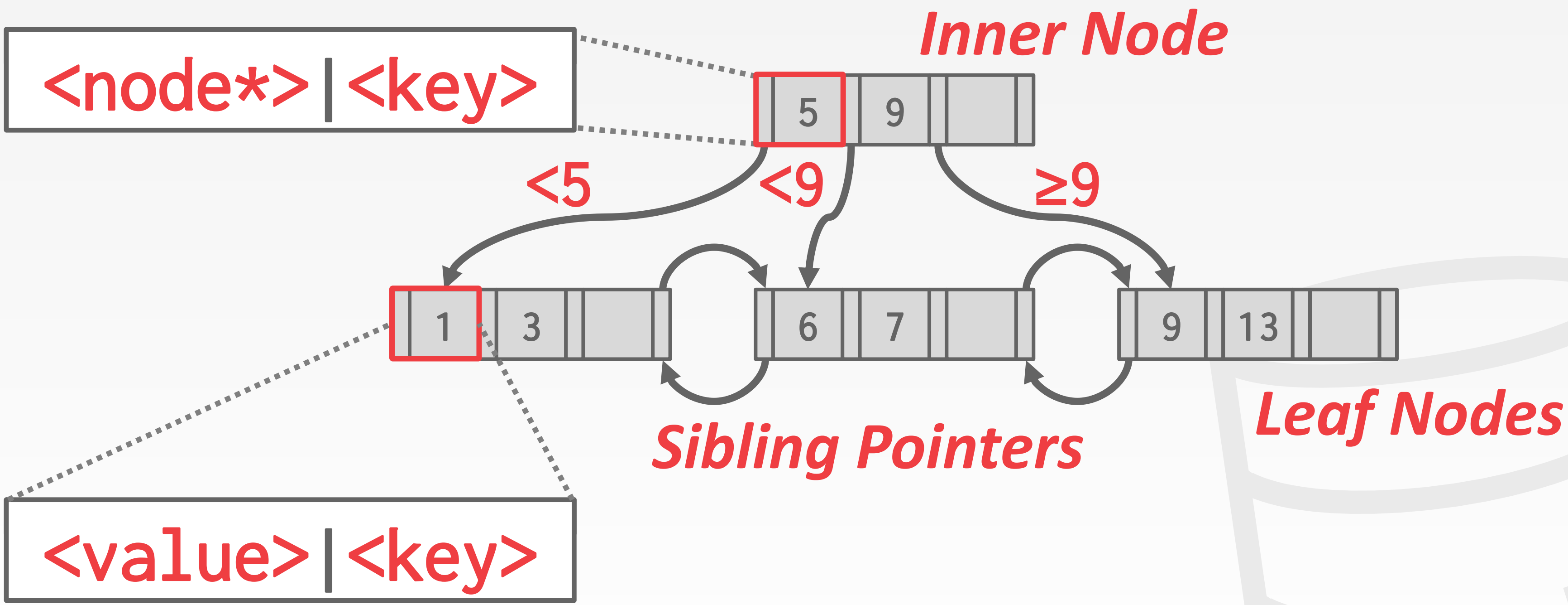
# B+TREE EXAMPLE



# B+TREE EXAMPLE



# B+TREE EXAMPLE



# NODES

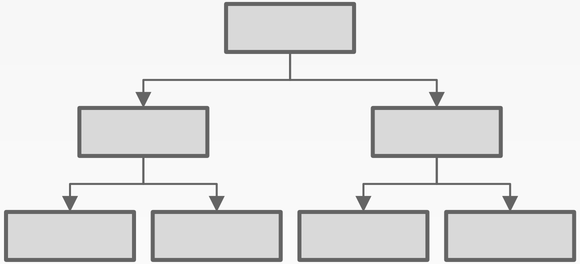
Every B+Tree node is comprised of an array of key/value pairs stored in a **page**.

→ The keys are derived from the attribute(s) that the index is based on.

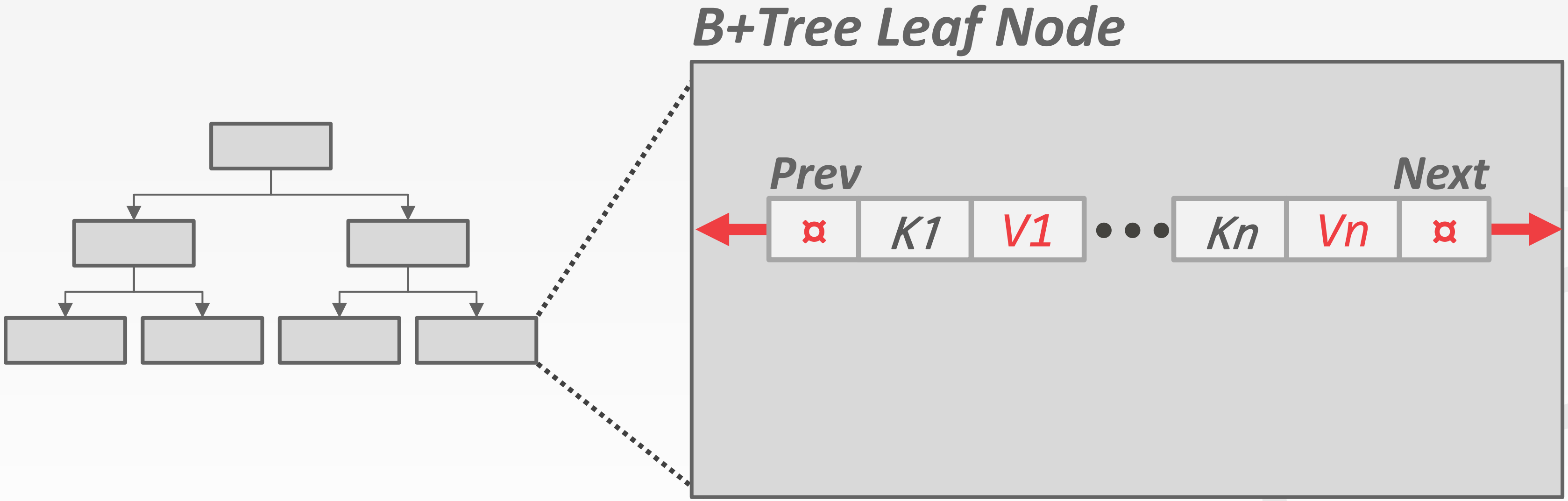
→ The values will differ based on whether the node is classified as an **inner node** or a **leaf node.**

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

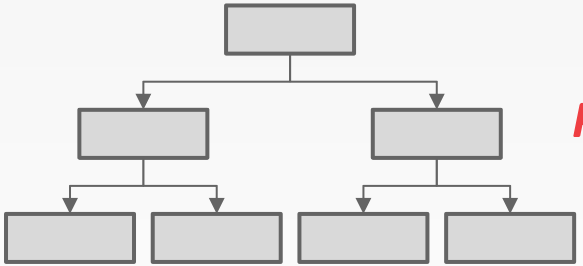
# B+TREE LEAF NODES

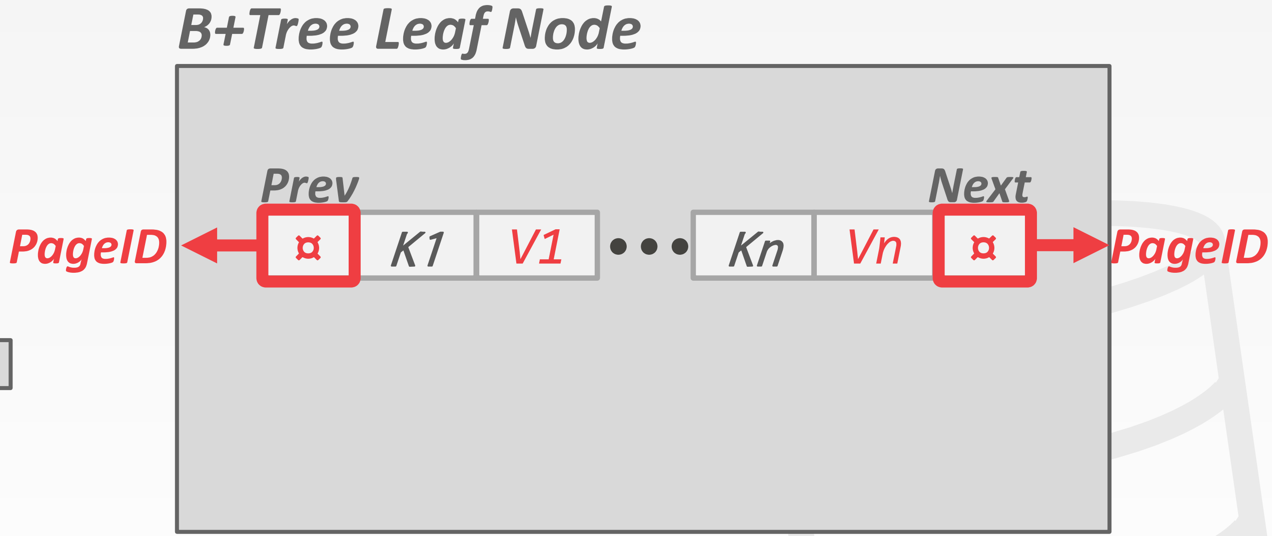


# B+TREE LEAF NODES

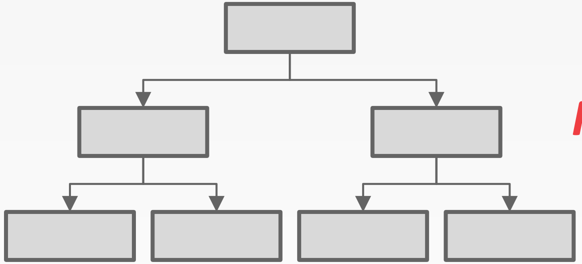


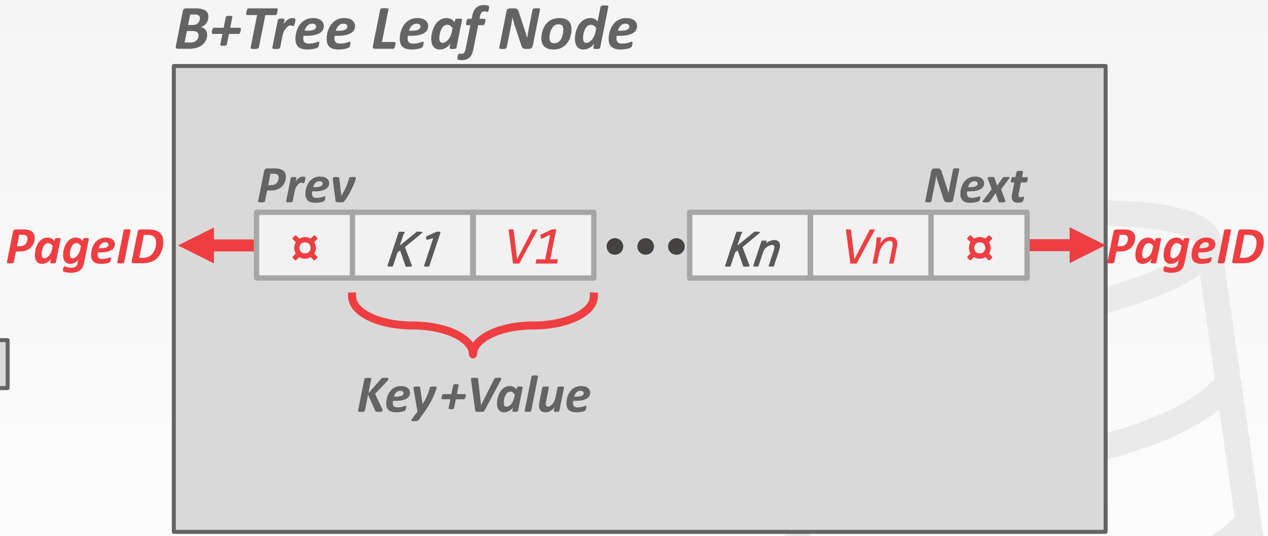
# B+TREE LEAF NODES



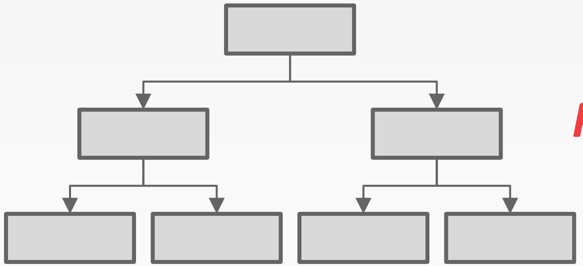


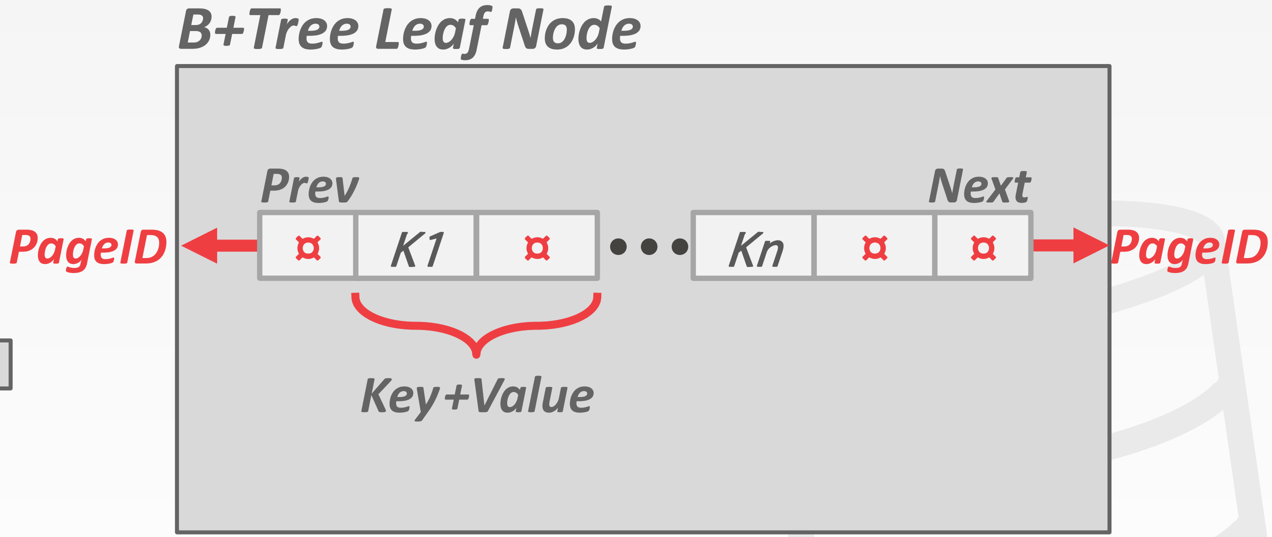
# B+TREE LEAF NODES



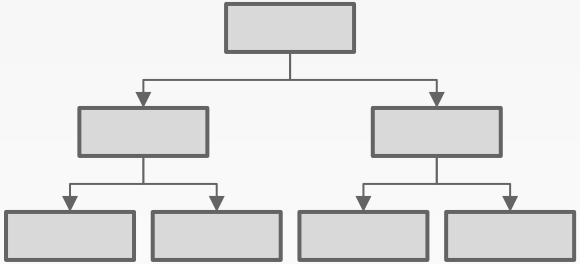


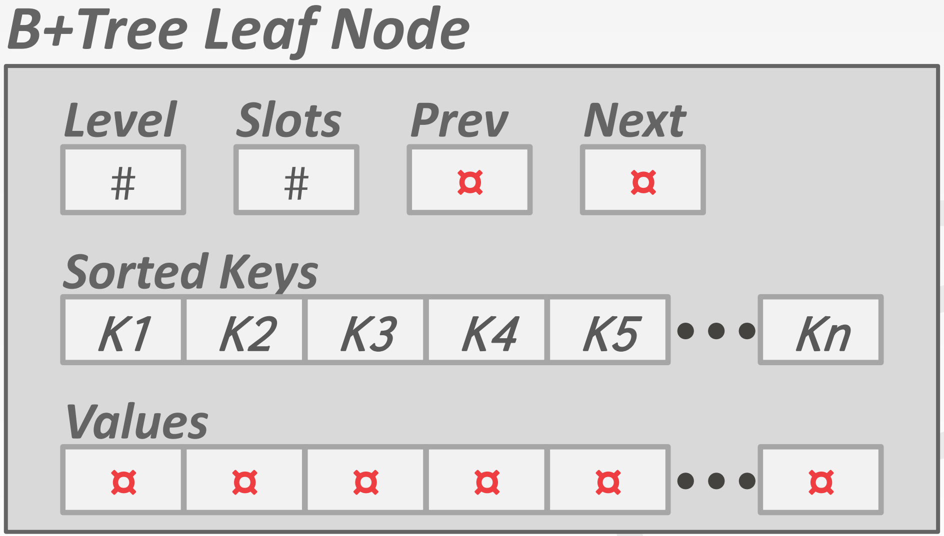
# B+TREE LEAF NODES



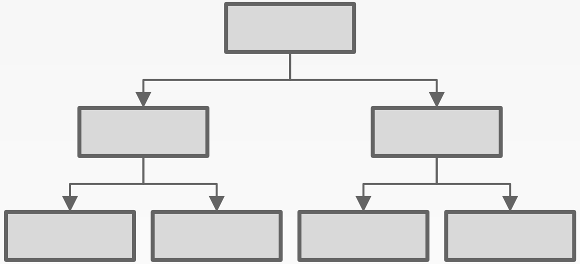


# B+TREE LEAF NODES

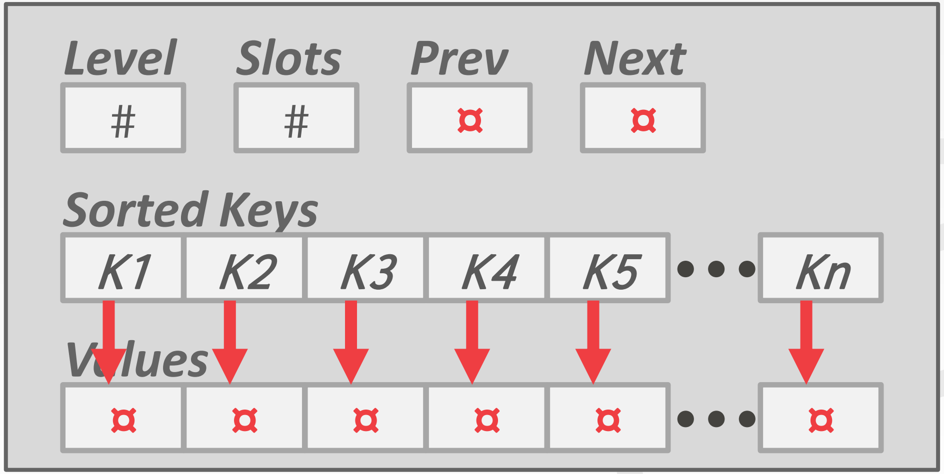




# B+TREE LEAF NODES



B+Tree Leaf Node



# LEAF NODE VALUES

**Approach #1: Record IDs**

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

**Approach #2: Tuple Data**

→ The leaf nodes (of the primary key index) store the actual contents of the tuple.

→ Secondary indexes must store the Record ID as their values.

# LEAF NODE VALUES

## Approach #1: Record IDs

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

## Approach #2: Tuple Data

→ The leaf nodes (of the primary key index) store the actual contents of the tuple.

→ Secondary indexes must store the Record ID as their values.

# B-TREE VS. B+TREE

The original **B-Tree** from 1972 stored keys and 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 node L.
Put data entry into L in sorted order.

If L has enough space, done!

Otherwise, (optionally, try to re-distribute a key to sibling nodes first) split L keys into L and a new node L2

→ Split entries evenly, copy up middle key.
→ Insert index entry pointing to L2 into parent of L.

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

Source: Chris Re

# B+TREE VISUALIZATION

**https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html**

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

# SELECTION CONDITIONS

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

Example: Index on <a,b,c>

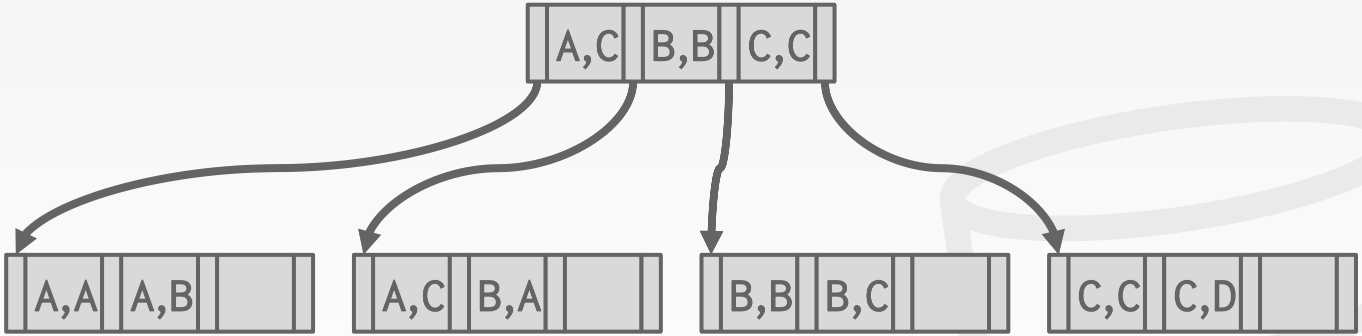
→ Supported: (a=5 AND b=3)

→ Not Supported: (b=3)

Not all DBMSs support this.

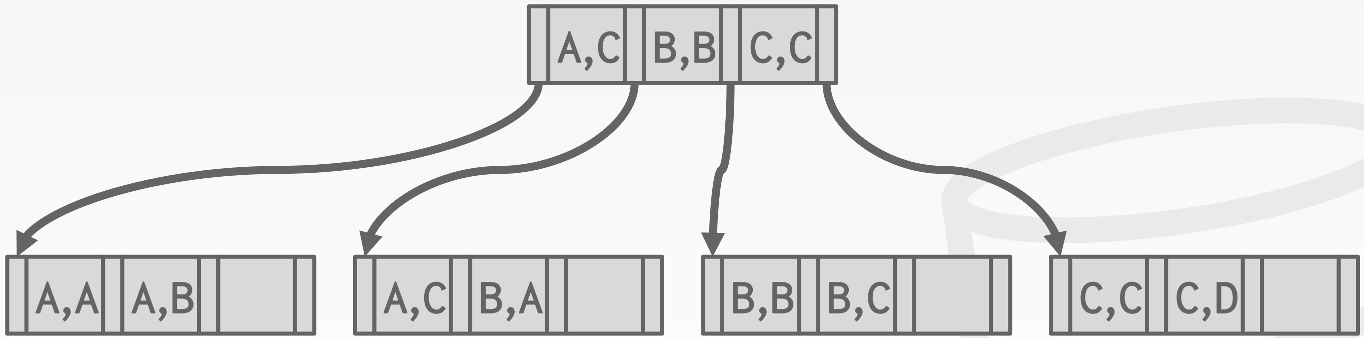
For a hash index, we must have all attributes in search key.

# SELECTION CONDITIONS



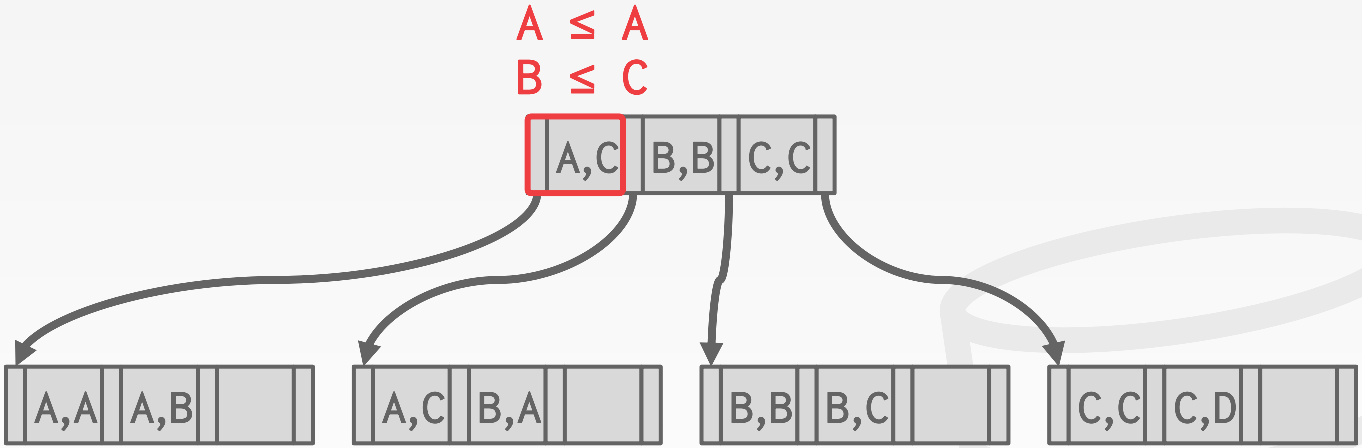
# SELECTION CONDITIONS

Find Key=(A,B)



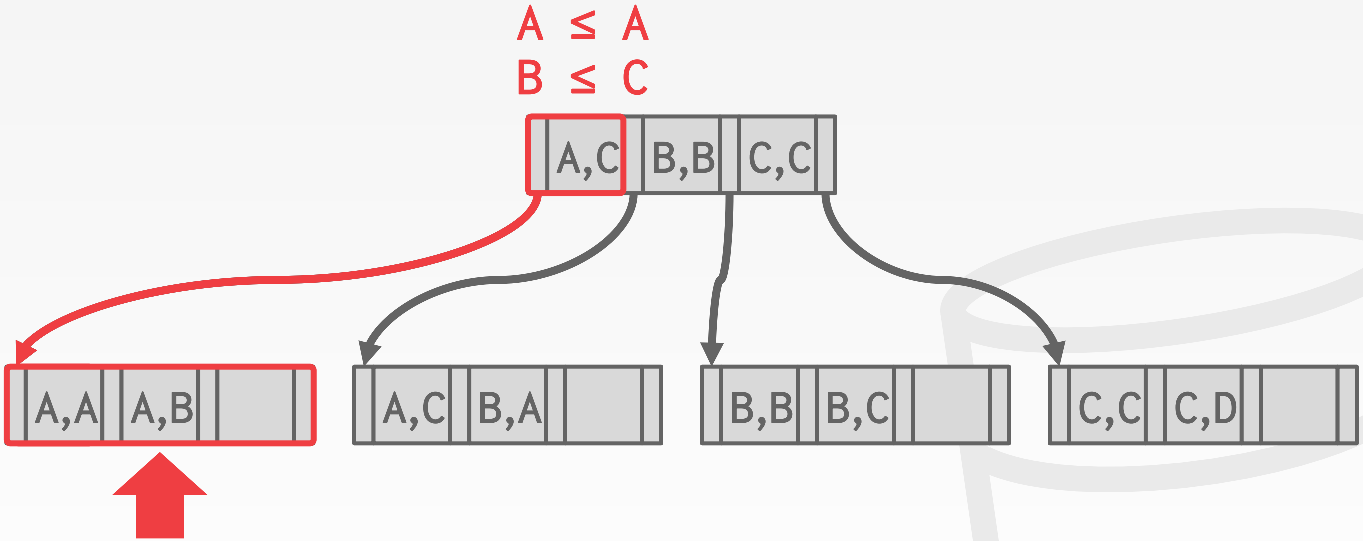
# SELECTION CONDITIONS

Find Key=(A,B)



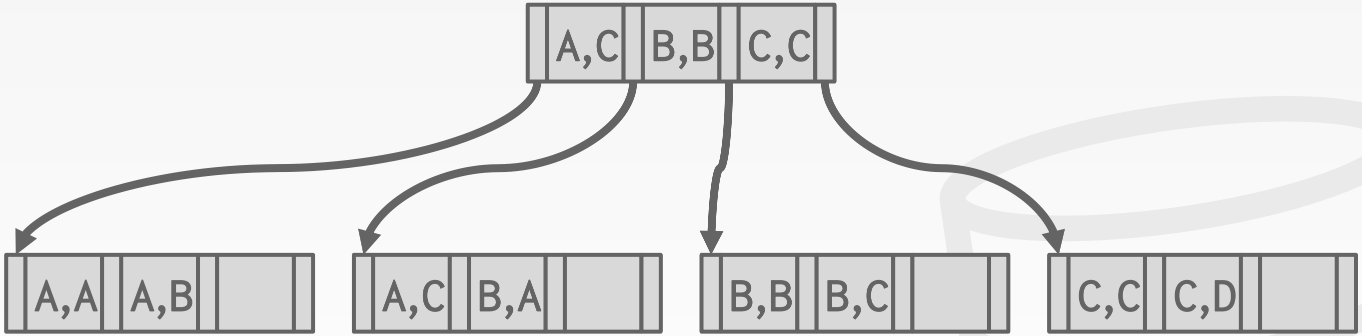
# SELECTION CONDITIONS

Find Key=(A,B)



# SELECTION CONDITIONS

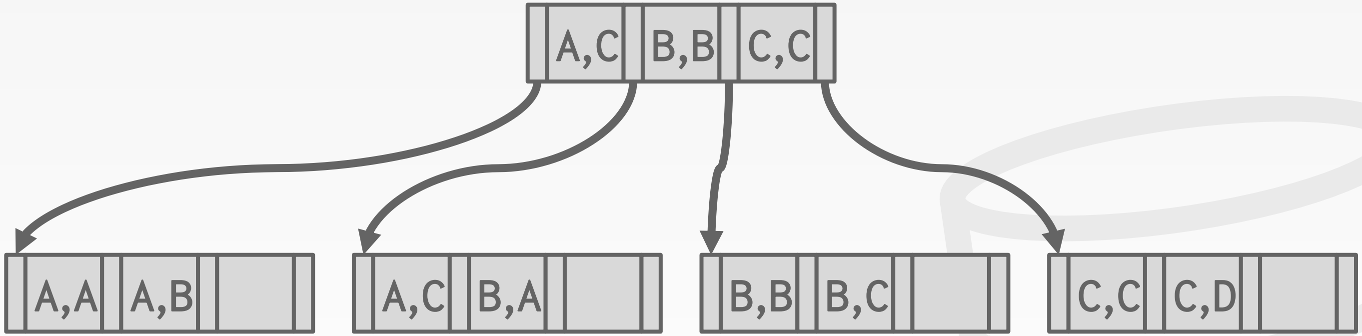
Find Key=(A,B)



# SELECTION CONDITIONS

Find Key=(A,B)

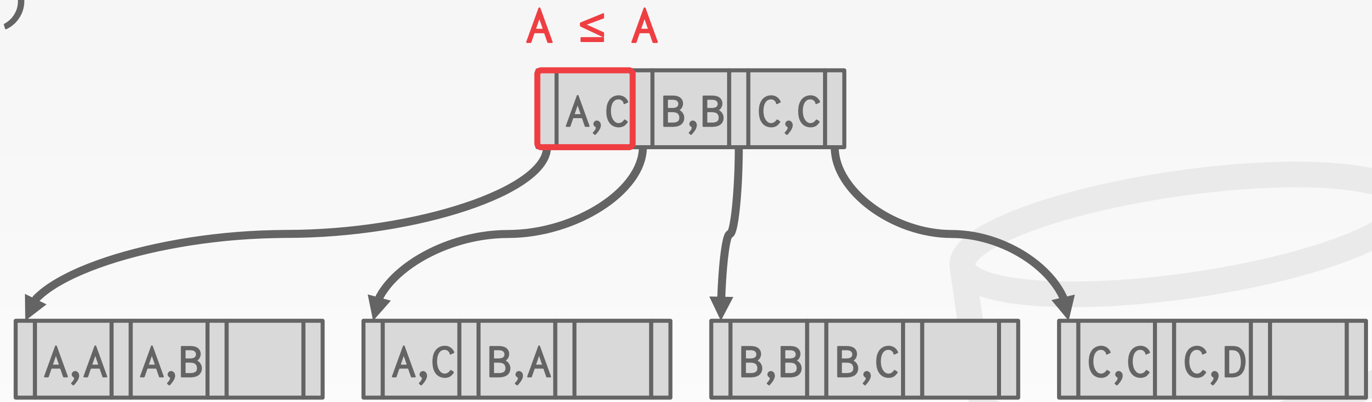
Find Key=(A,*)



# SELECTION CONDITIONS

Find Key=(A,B)

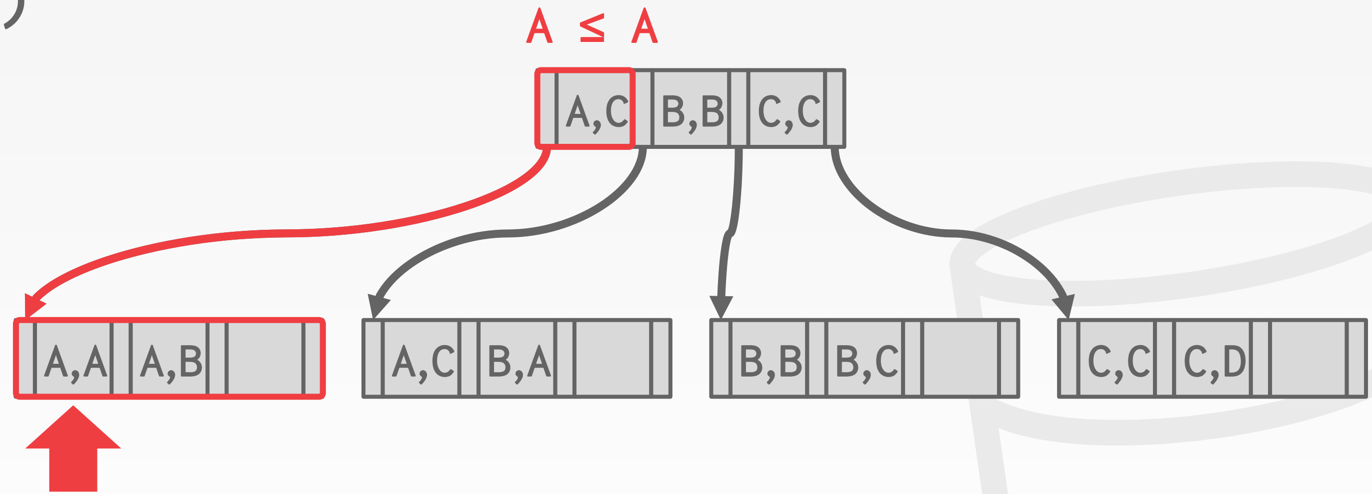
Find Key=(A,*)



# SELECTION CONDITIONS

Find Key=(A,B)

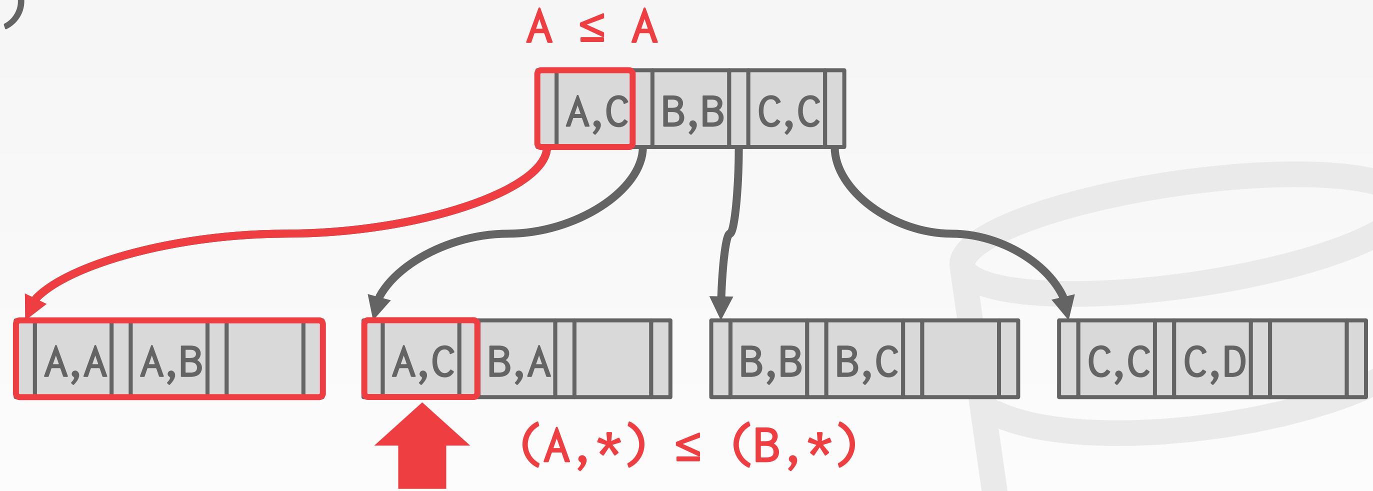
Find Key=(A,*)



# SELECTION CONDITIONS

Find Key=(A,B)

Find Key=(A,*)



# B+TREE – 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.

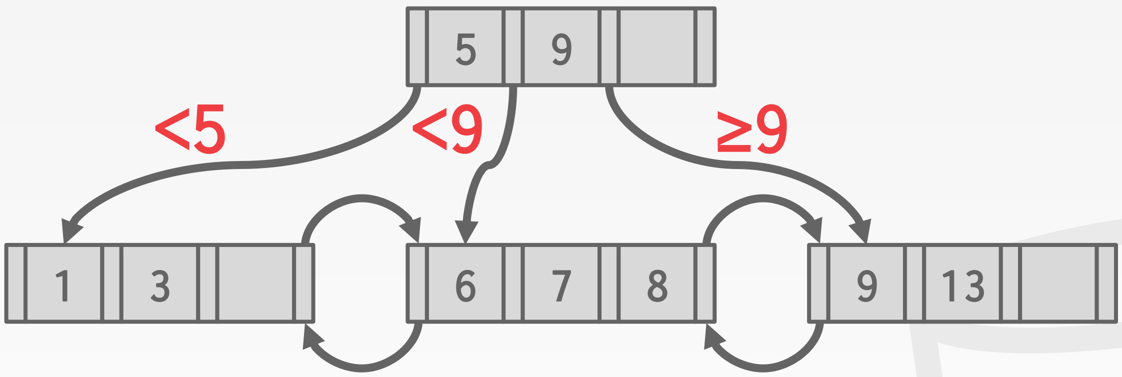
→ The DBMS can still use partial keys to find tuples.

## Approach #2: Overflow Leaf Nodes

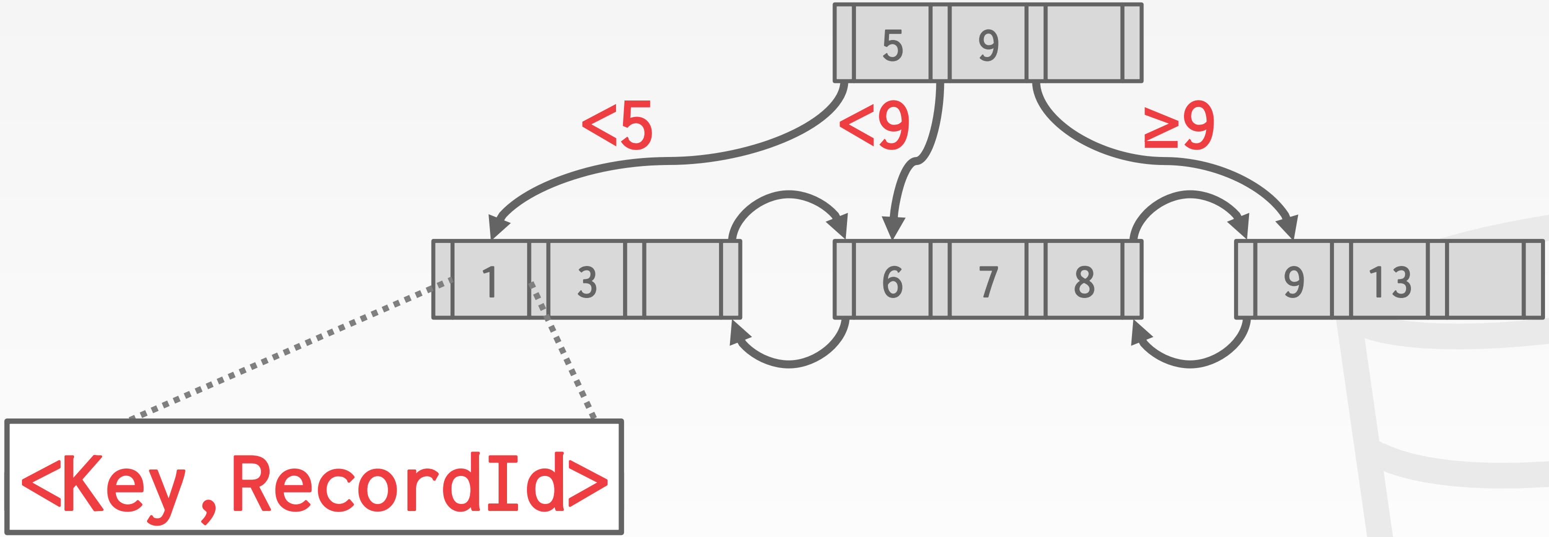
→ Allow leaf nodes to spill into overflow nodes that contain the duplicate keys.

→ This is more complex to maintain and modify.

# B+TREE – APPEND RECORD ID

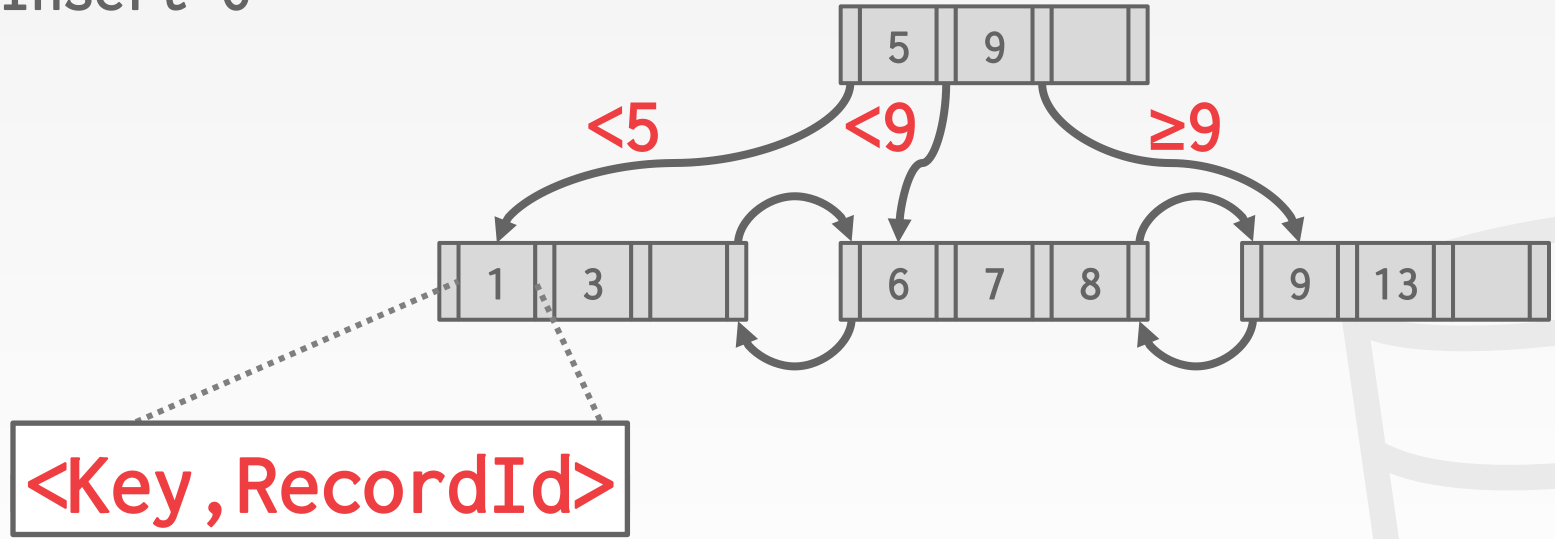


# B+TREE – APPEND RECORD ID



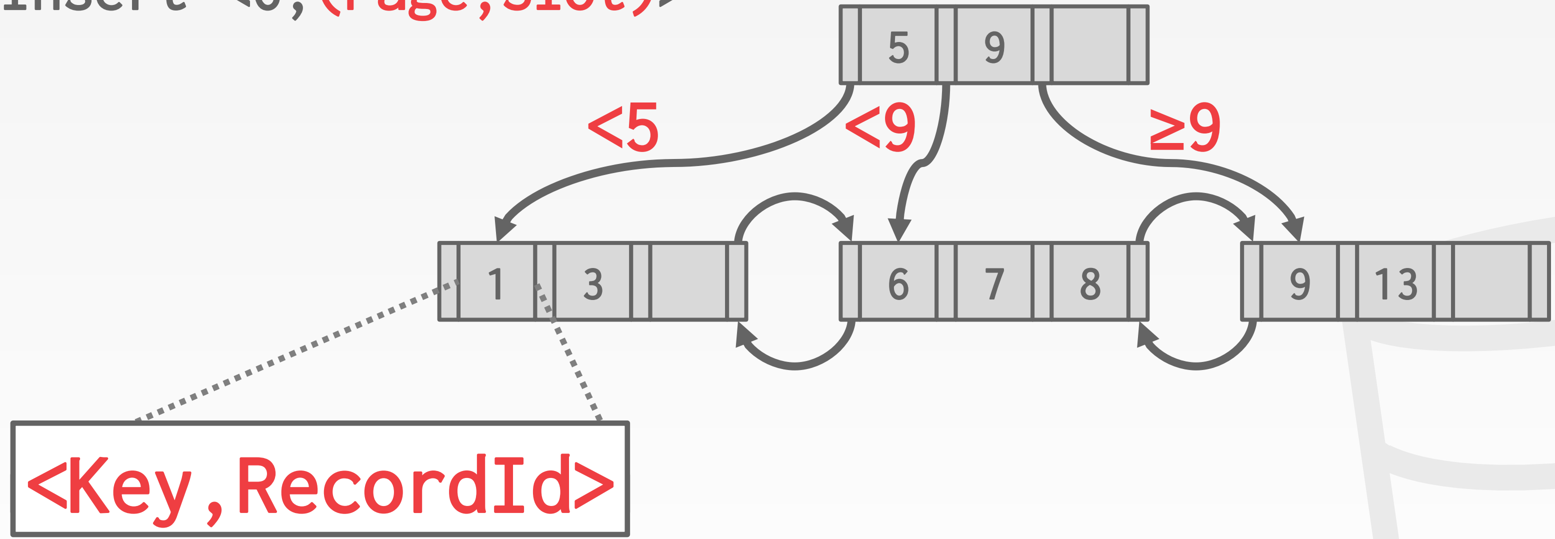
# B+TREE – APPEND RECORD ID

Insert 6



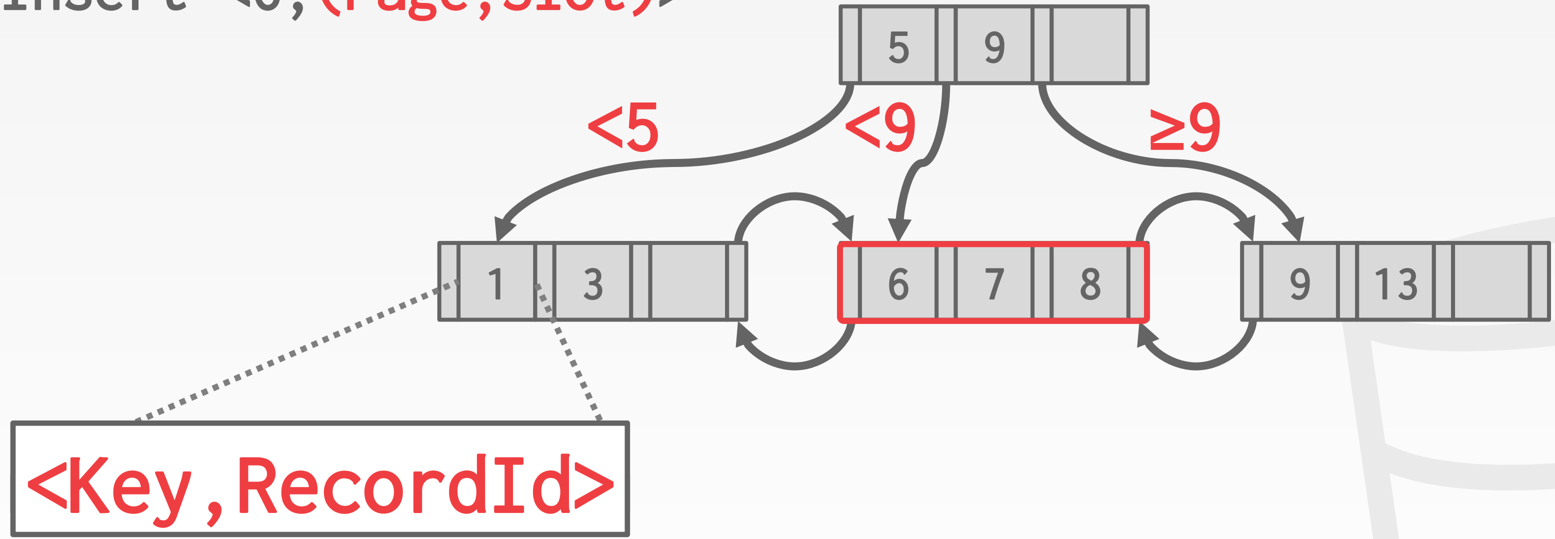
# B+TREE – APPEND RECORD ID

Insert <6,(Page,Slot)>



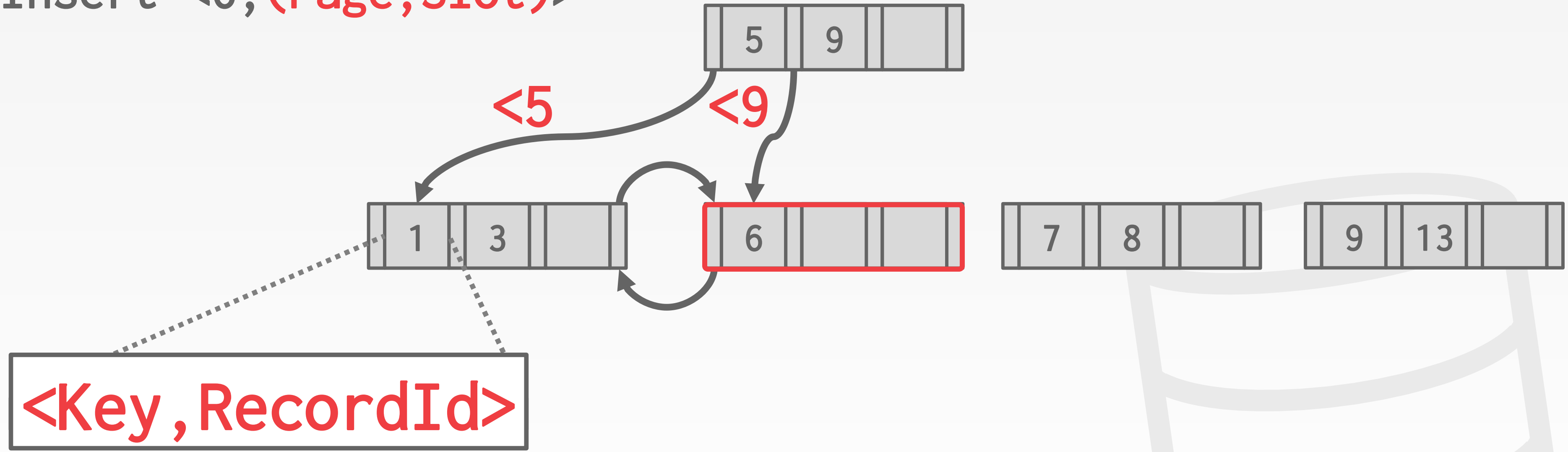
# B+TREE – APPEND RECORD ID

Insert <6,(Page,Slot)>



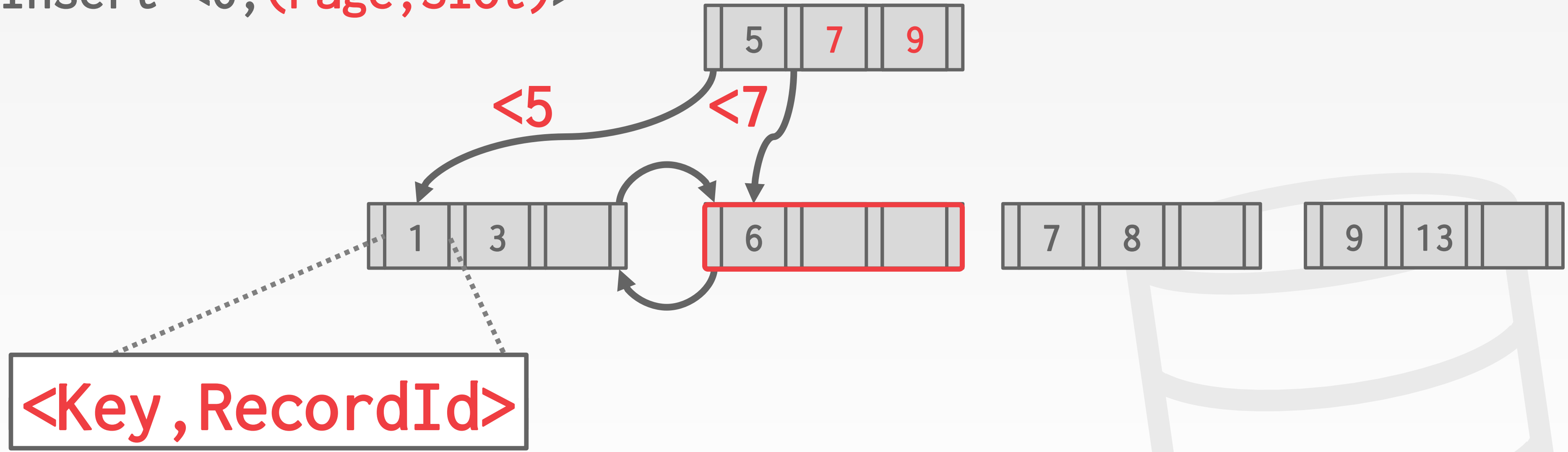
# B+TREE – APPEND RECORD ID

Insert <6,(Page,Slot)>



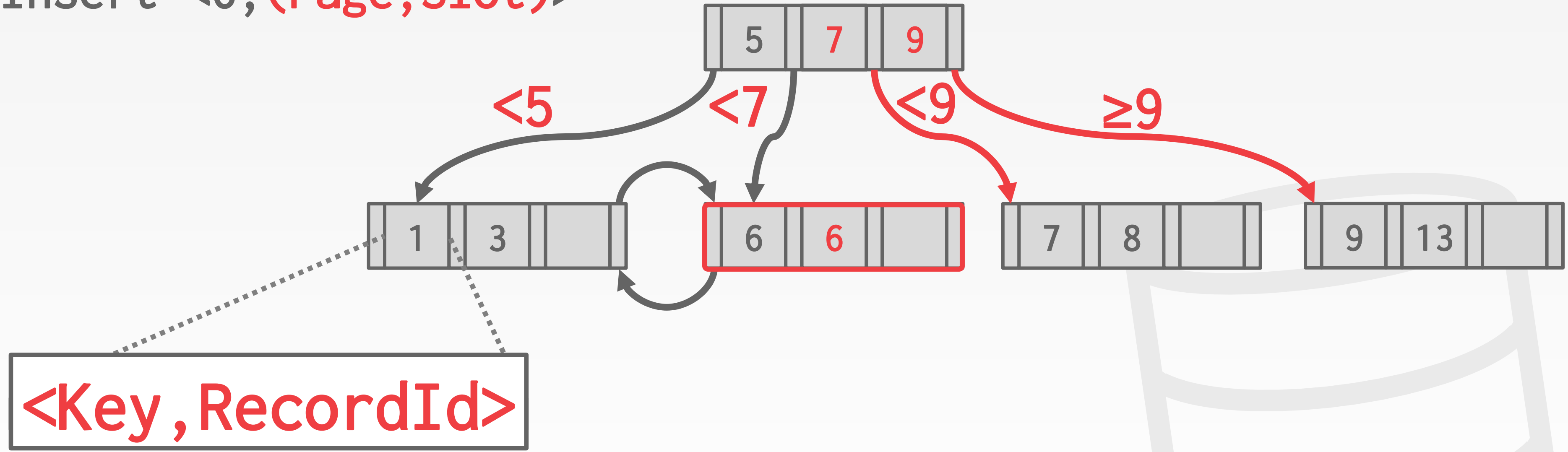
# B+TREE – APPEND RECORD ID

Insert <6,(Page,Slot)>



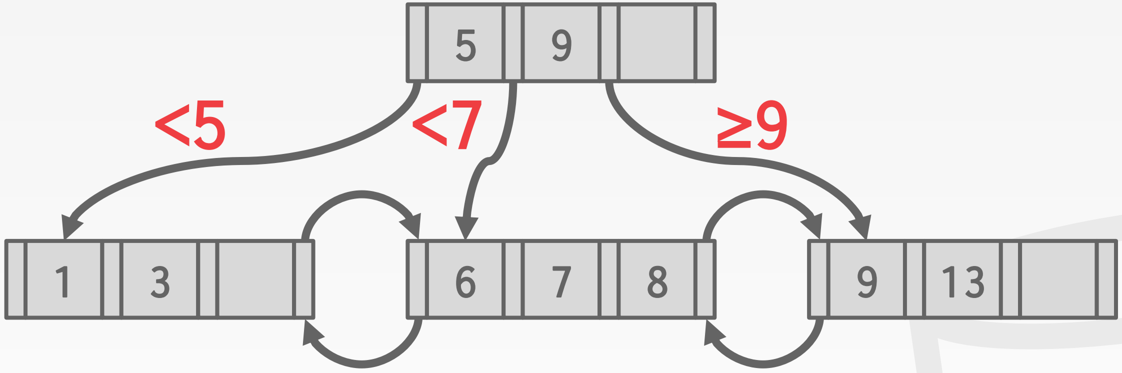
# B+TREE – APPEND RECORD ID

Insert <6,(Page,Slot)>



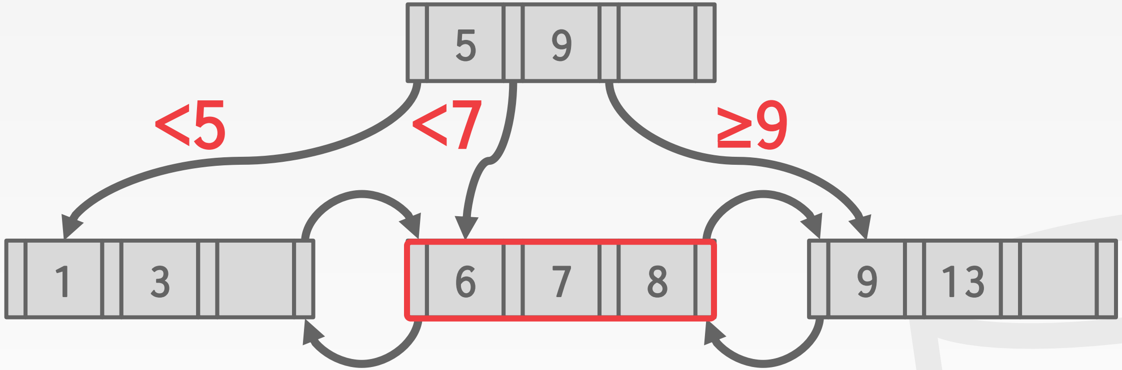
# B+TREE – OVERFLOW LEAF NODES

Insert 6



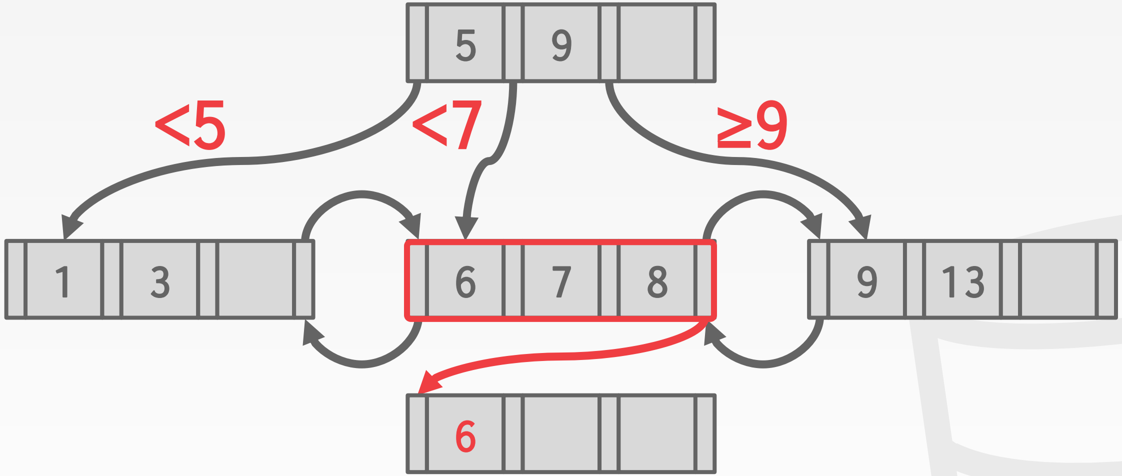
# B+TREE – OVERFLOW LEAF NODES

Insert 6



# B+TREE – OVERFLOW LEAF NODES

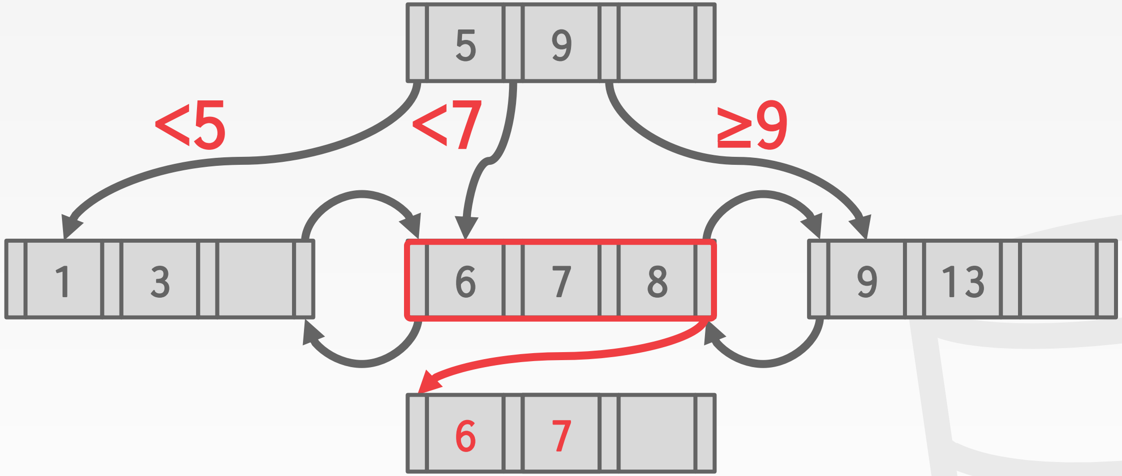
Insert 6



# B+TREE – OVERFLOW LEAF NODES

Insert 6

Insert 7

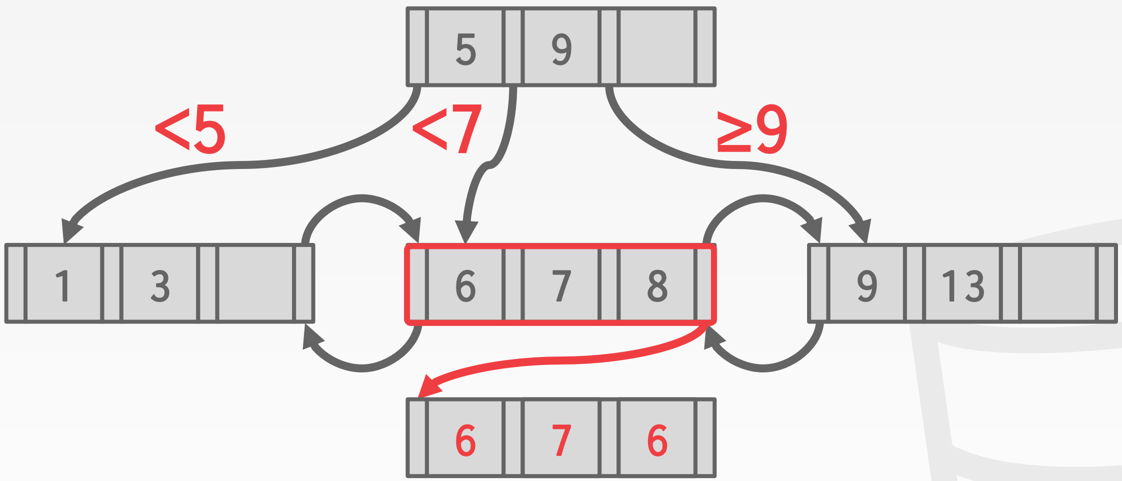


# B+TREE – OVERFLOW LEAF NODES

Insert 6

Insert 7

Insert 6



# 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.

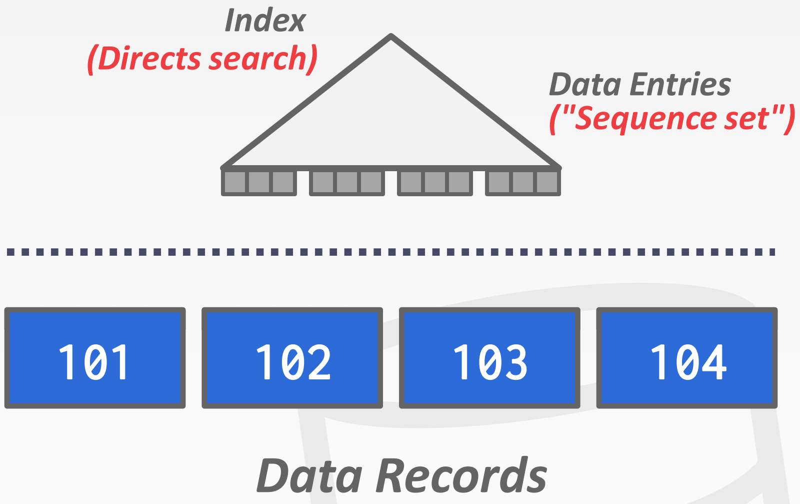
→ If a table does not contain a primary key, the DBMS will automatically make a hidden primary key.

Other DBMSs cannot use them at all.

# CLUSTERED B+TREE

Traverse to the left-most leaf page and then retrieve tuples from all leaf pages.

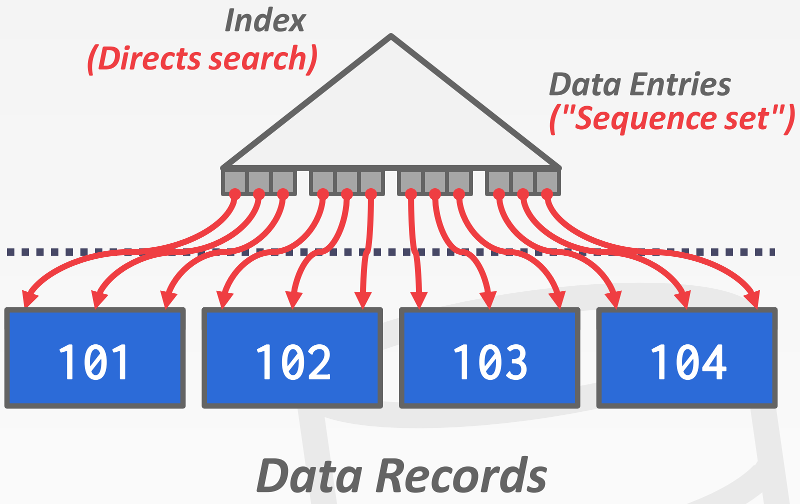
This will always be better than external sorting.



# CLUSTERED B+TREE

Traverse to the left-most leaf page and then retrieve tuples from all leaf pages.

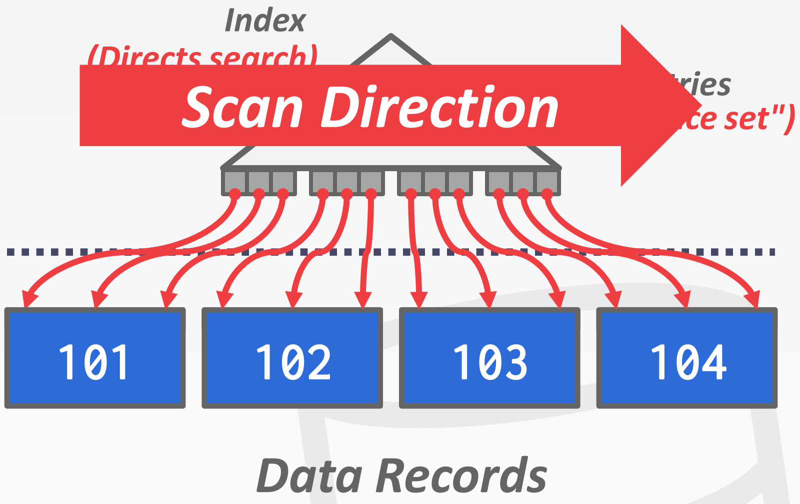
This will always be better than external sorting.



# CLUSTERED B+TREE

Traverse to the left-most leaf page and then retrieve tuples from all leaf pages.

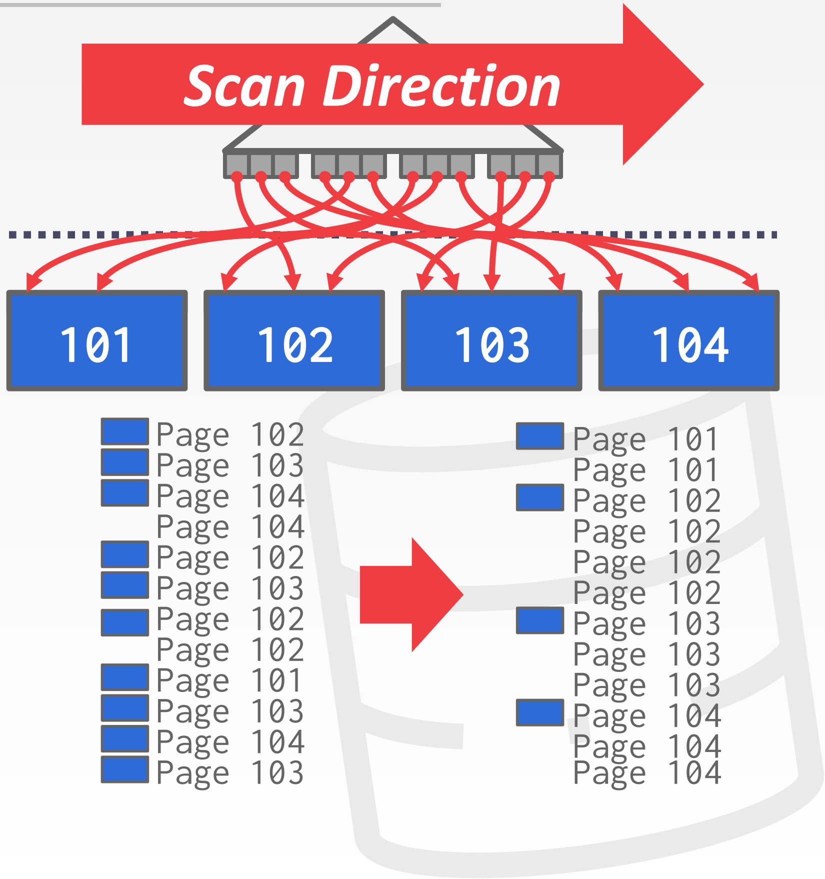
This will always be better than external sorting.



# INDEX SCAN PAGE SORTING

Retrieving tuples in the order they appear in a non-clustered index can be very inefficient.

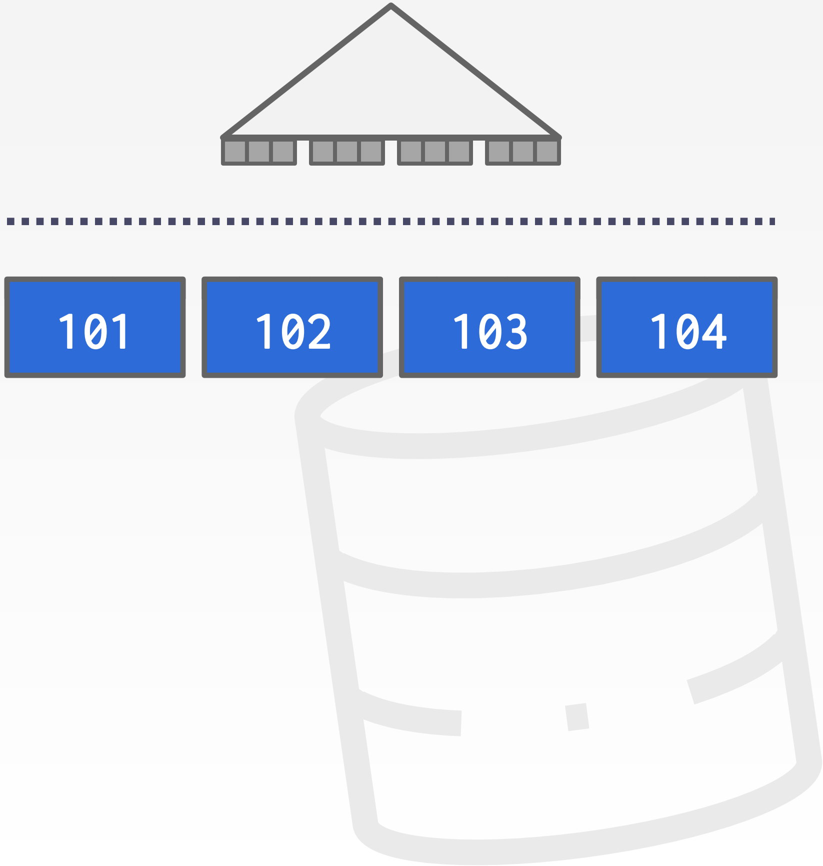
The DBMS can first figure out all the tuples that it needs and then sort them based on their Page ID.



# INDEX SCAN PAGE SORTING

Retrieving tuples in the order they appear in a non-clustered index can be very inefficient.

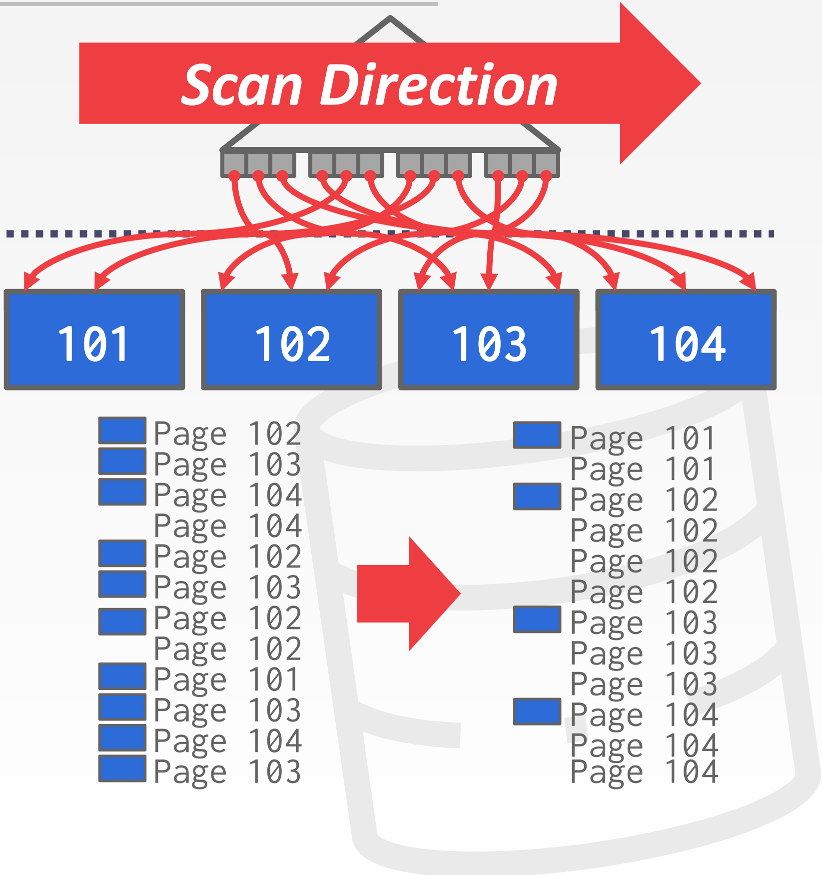
The DBMS can first figure out all the tuples that it needs and then sort them based on their Page ID.



# INDEX SCAN PAGE SORTING

Retrieving tuples in the order they appear in a non-clustered index can be very inefficient.

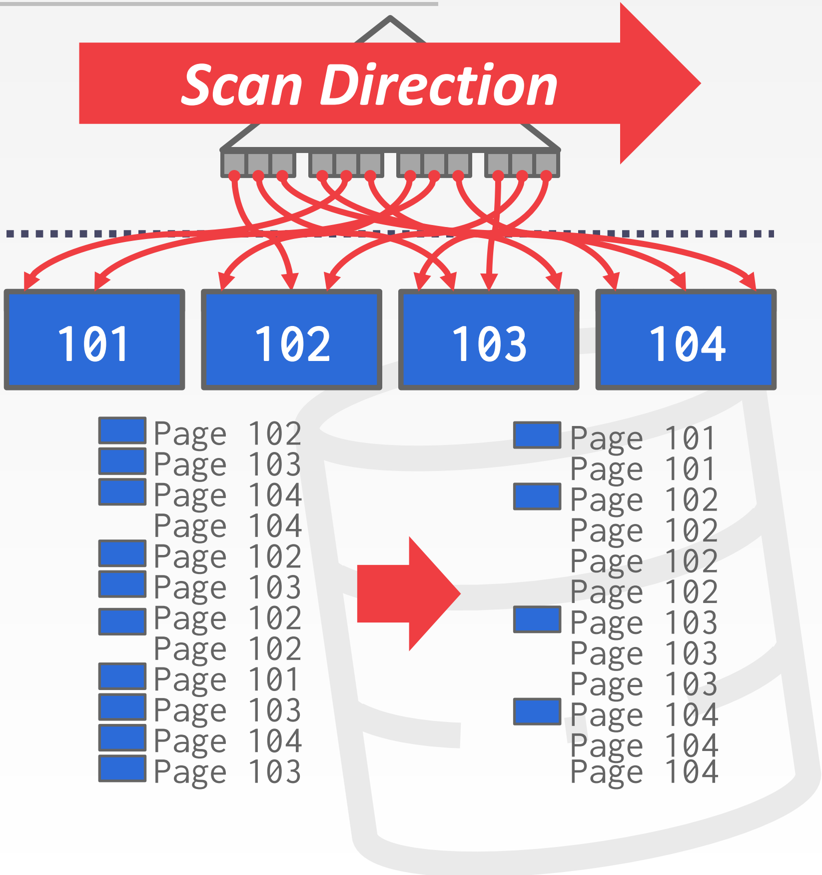
The DBMS can first figure out all the tuples that it needs and then sort them based on their Page ID.



# INDEX SCAN PAGE SORTING

Retrieving tuples in the order they appear in a non-clustered index can be very inefficient.

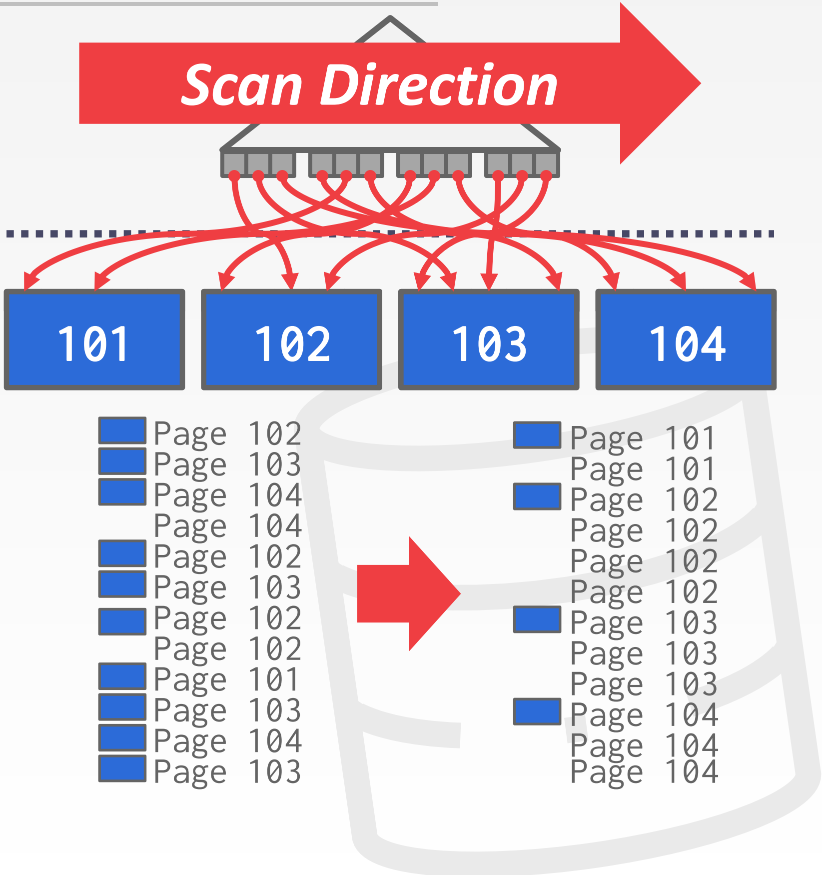
The DBMS can first figure out all the tuples that it needs and then sort them based on their Page ID.



# INDEX SCAN PAGE SORTING

Retrieving tuples in the order they appear in a non-clustered index can be very inefficient.

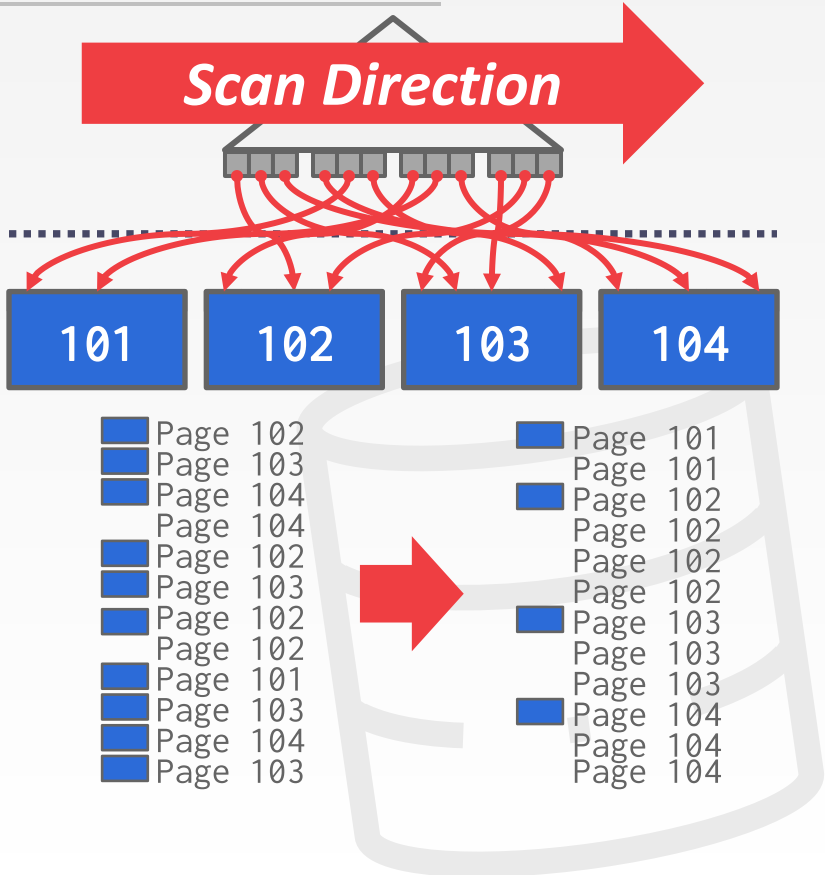
The DBMS can first figure out all the tuples that it needs and then sort them based on their Page ID.



# INDEX SCAN PAGE SORTING

Retrieving tuples in the order they appear in a non-clustered index can be very inefficient.

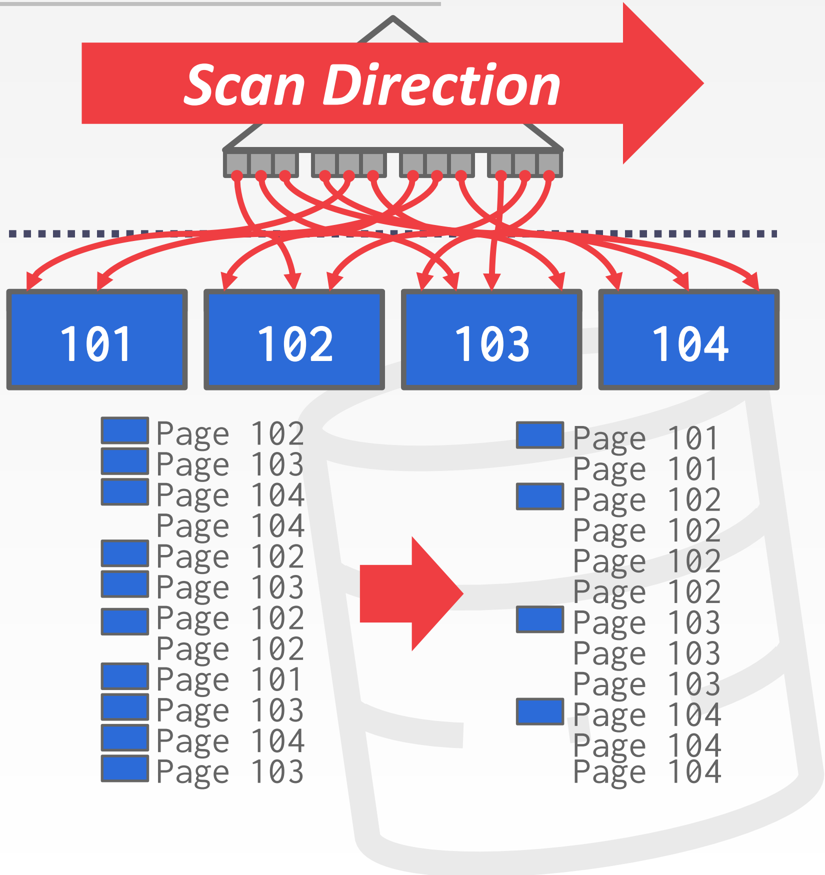
The DBMS can first figure out all the tuples that it needs and then sort them based on their Page ID.



# INDEX SCAN PAGE SORTING

Retrieving tuples in the order they appear in a non-clustered index can be very inefficient.

The DBMS can first figure out all the tuples that it needs and then sort them based on their Page ID.



# DEMO

B+Tree vs. Hash Indexes

Table Clustering

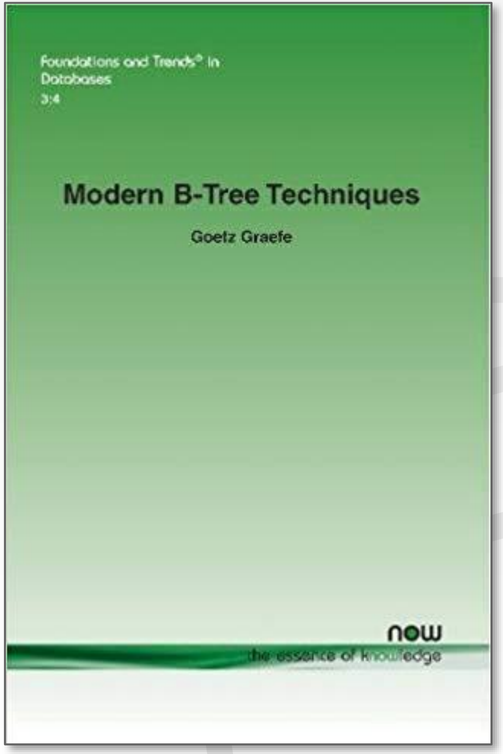
# B+TREE DESIGN CHOICES

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 they are half full.

Delaying a merge operation may reduce the amount of reorganization.

It may also be better to just let smaller nodes 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.

# INTRA-NODE SEARCH

**Approach #1: Linear**

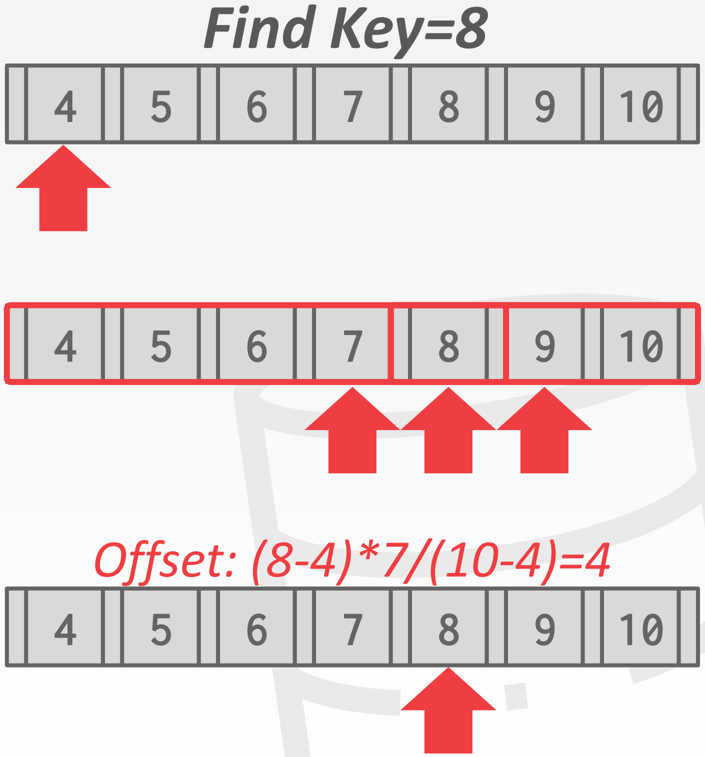
→ Scan node keys from beginning to end.

**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

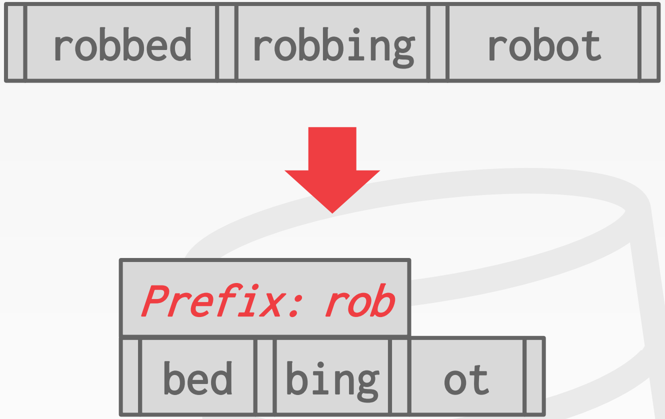
Bulk Insert

Many more…

# PREFIX COMPRESSION

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

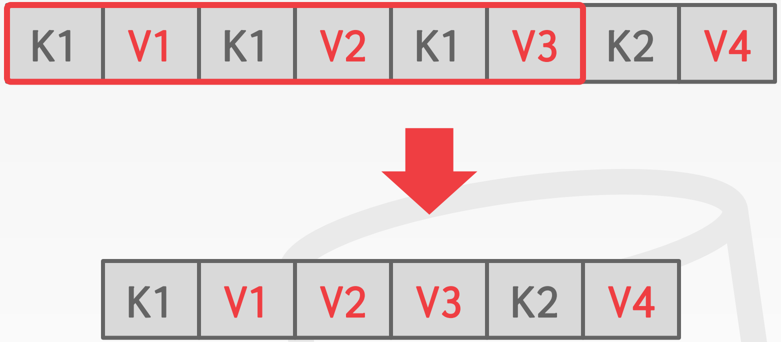
Instead of storing the entire key each time, extract common prefix and store only unique suffix for each key.
→ Many variations.



# DEDUPLICATION

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

The leaf node can store the key once and then maintain a list of tuples with that key (similar to what we discussed for hash tables).

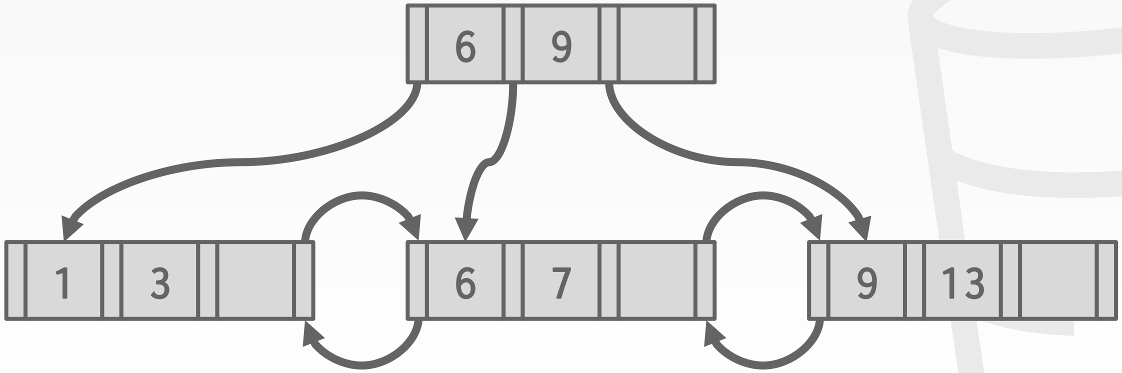


# BULK INSERT

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**



# CONCLUSION

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

# NEXT CLASS

Sorting & Aggregations