Intro to Databases (COMP_SCI 339)

06 B+Tree Index

Northwestern University

WINTER 2024

Andrew Crotty

ADMINISTRIVIA

Project #2 is due Sunday 2/4 @ 11:59pm

Exam #1 will be on 1/29 from 3:30-4:50pm

EXAM #1

Who: You

What: Exam #1

Where: Here

When: Monday 1/29 from 3:30-4:50pm

What to bring:

→ Pencil or pen with dark-colored ink

→ **<u>One</u>** double-sided 8.5x11" page of handwritten notes

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 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

Design Choices

Optimizations

B-TREE FAMILY

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

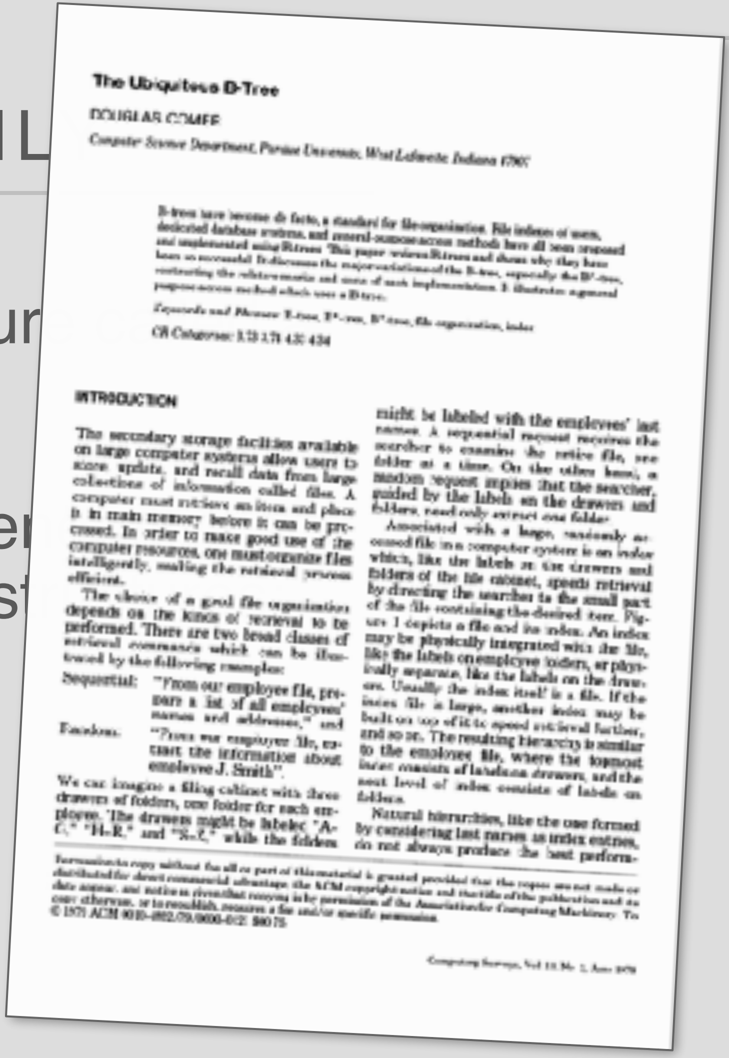
People also use the term to generally refer to a class of balanced tree data structures:

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

B-TREE FAMIL

There is a specific data structur
B-Tree.

People also use the term to gen
a class of balanced tree data st
→ **B-Tree** (1971)
→ **B+Tree** (1973)
→ **B*Tree** (1977?)
→ **B**$^{\textbf{link}}$**-Tree** (1981)


The Ubiquitous B-Tree

DOUGLAS COMER

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

INTRODUCTION

B+TREE

A **B+Tree** is a self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions always 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.

B+TREE PROPERTIES

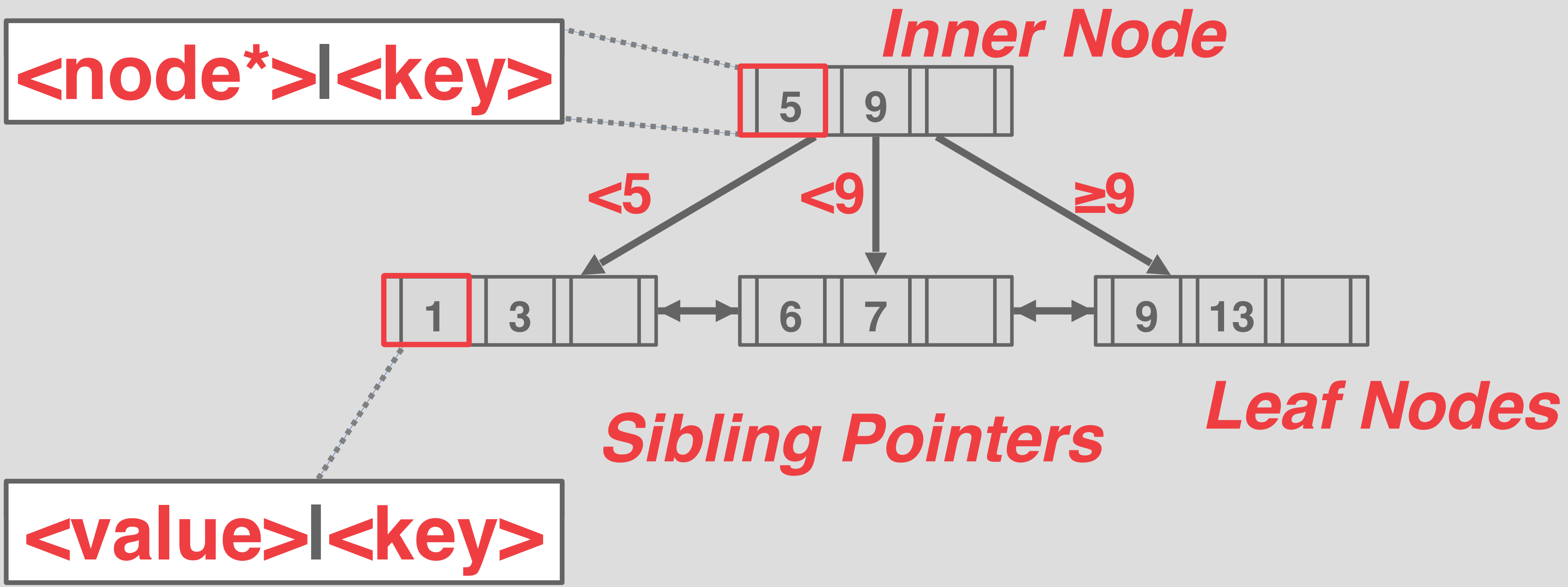
A B+Tree is an ***M***-way search tree with the following properties:

→ 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



NODES

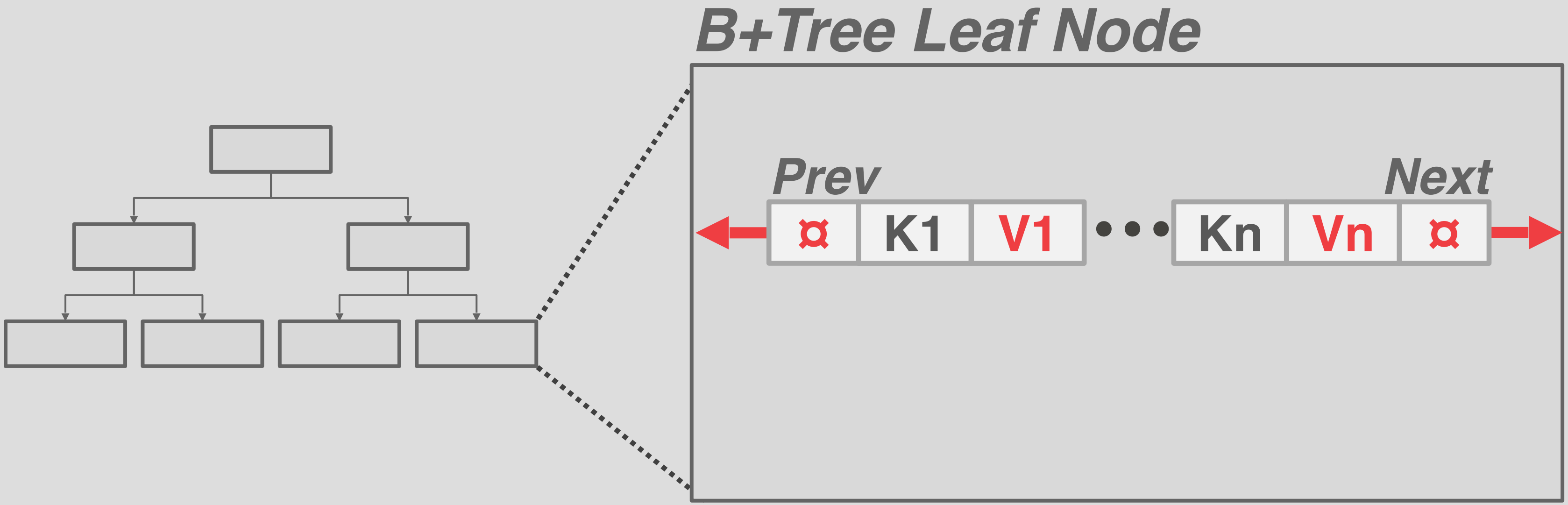
Every B+Tree node is comprised of an array of key/value pairs.

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

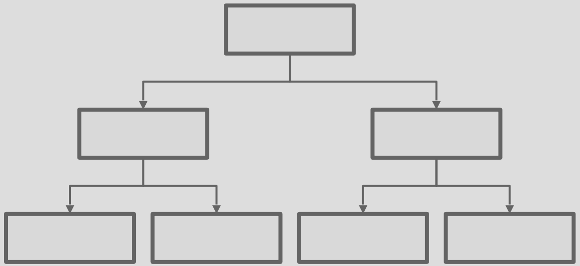
→ The values will differ based on whether the node is classified as an **inner node** or a **leaf node.**

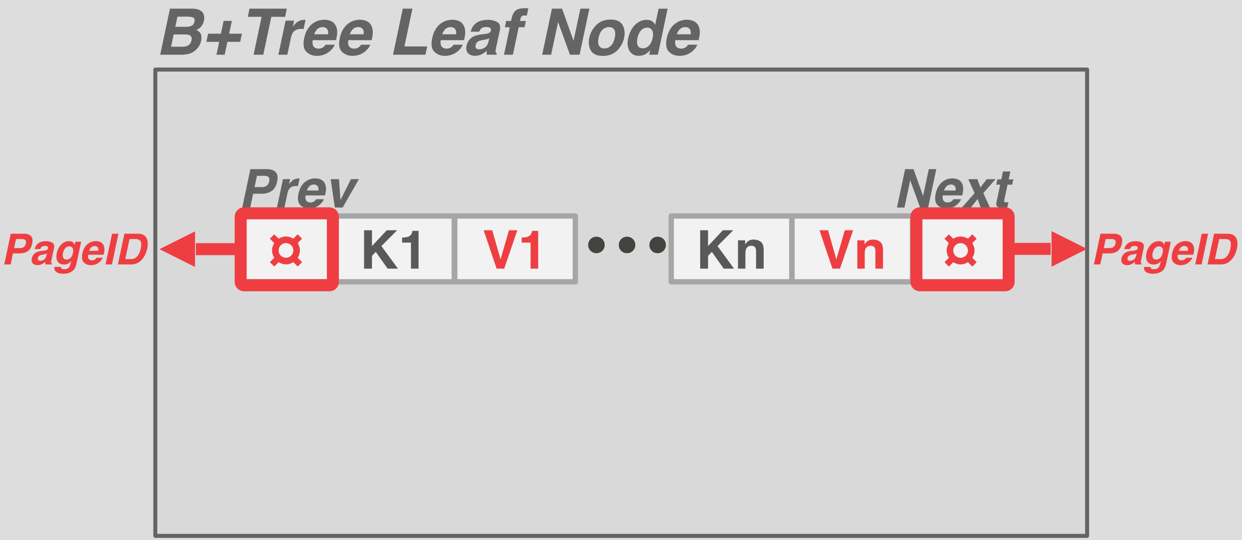
The arrays are (usually) kept in sorted key order.

B+TREE LEAF NODES

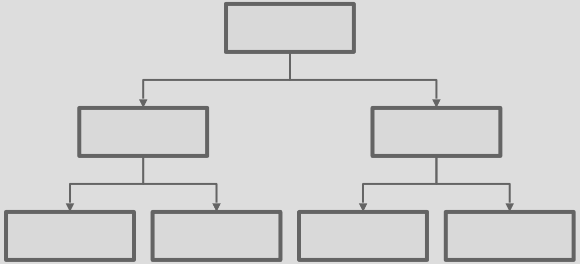


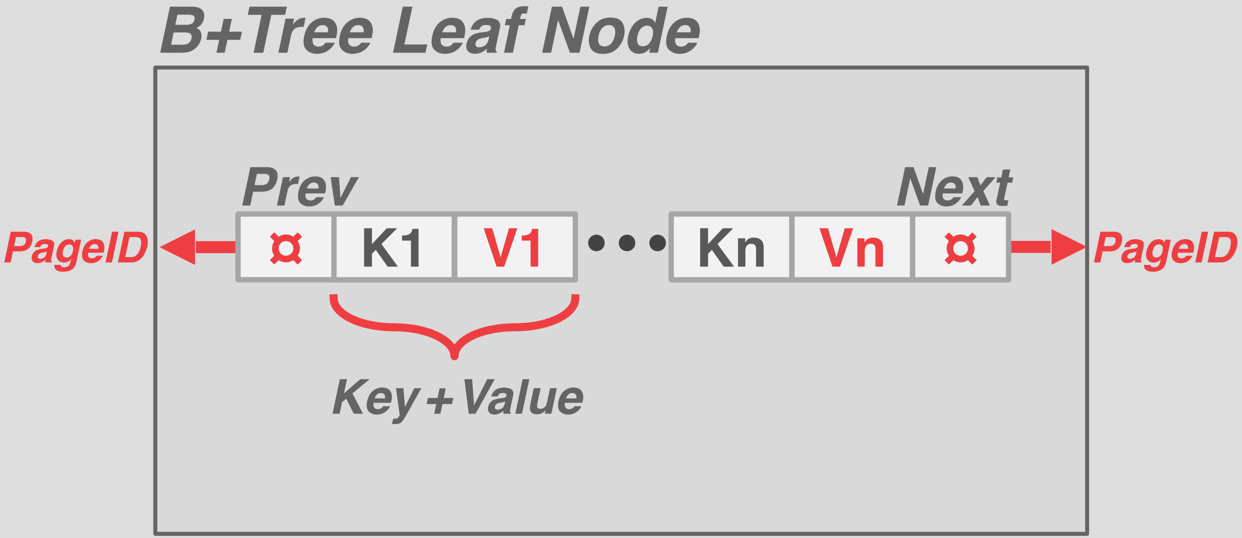
B+TREE LEAF NODES



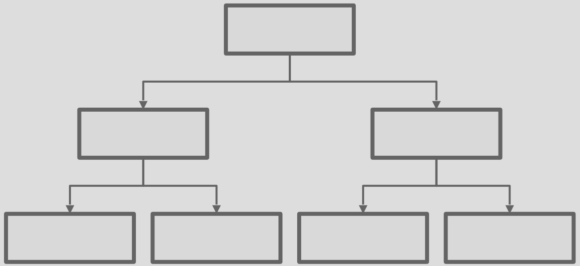


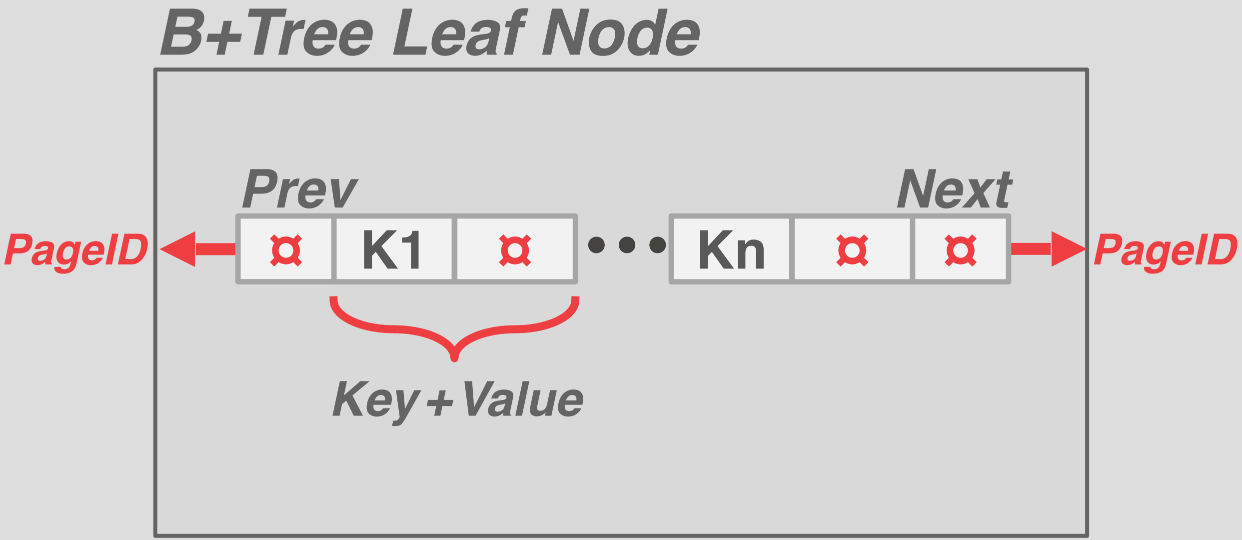
B+TREE LEAF NODES



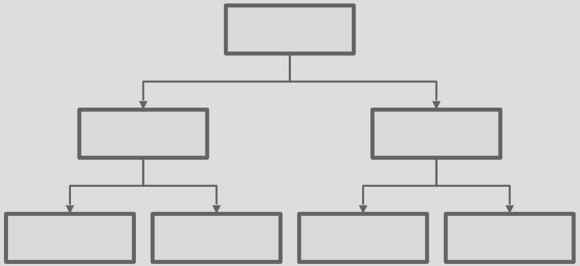


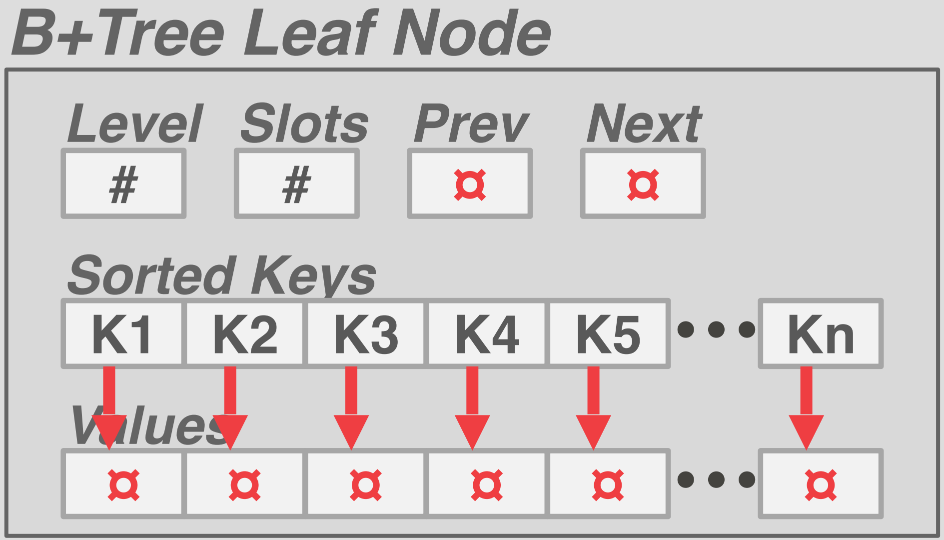
B+TREE LEAF NODES





B+TREE LEAF NODES





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 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**.
Insert data entry into **L** in sorted order.

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

Otherwise, split **L** keys 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 – 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+TREE VISUALIZATION

https://cmudb.io/btree

Source: David Gales (Univ. of San Francisco)

SELECTION CONDITIONS

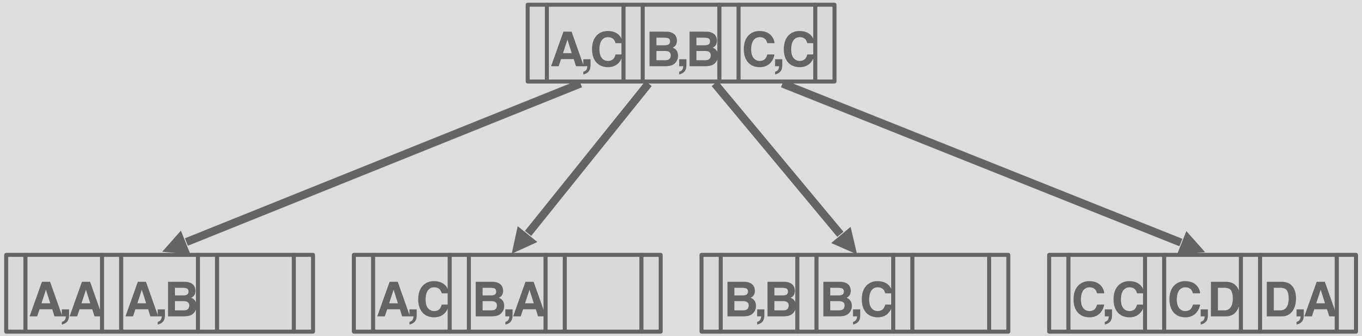
The DBMS can use a B+Tree index if the query provides any of the attributes of the search key.

Example: Index on **<a,b,c>**
→ Supported **(a=1 AND b=2 AND c=3)**
→ Supported: **(a=1 AND b=2)**
→ Supported: **(b=2)**, **(c=3)**

Not all DBMSs support this.

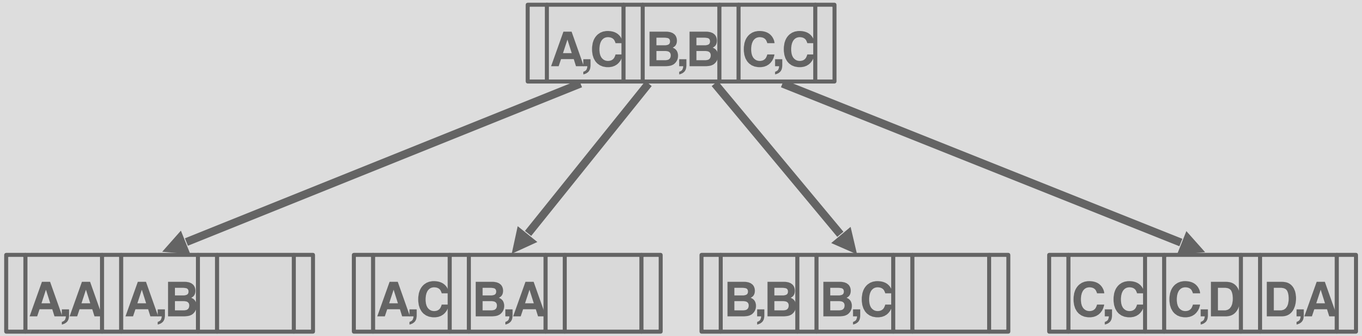
For a hash index, we must have all attributes in search key.

SELECTION CONDITIONS



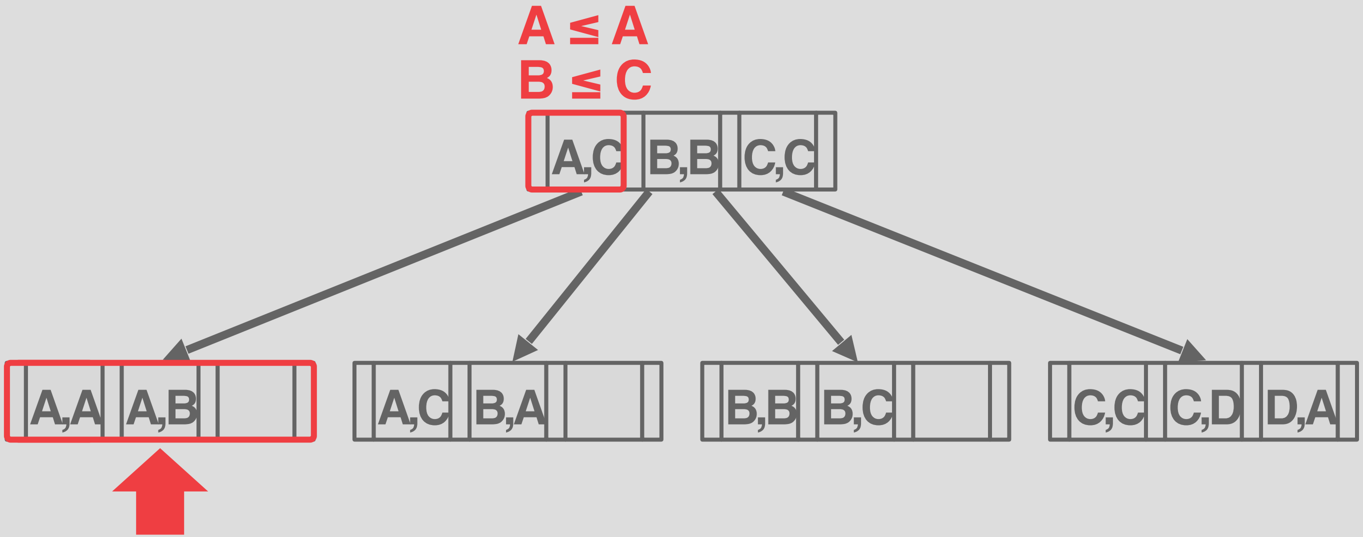
SELECTION CONDITIONS

Find Key=(A,B)



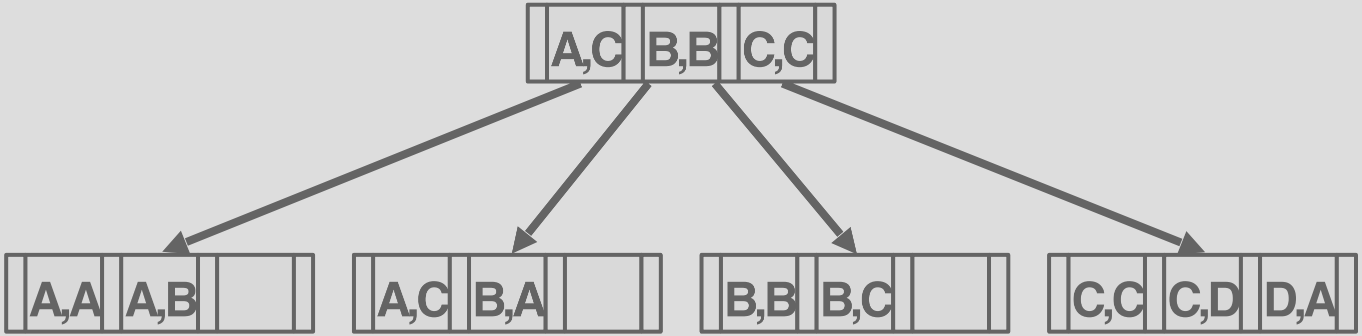
SELECTION CONDITIONS

Find Key=(A,B)



SELECTION CONDITIONS

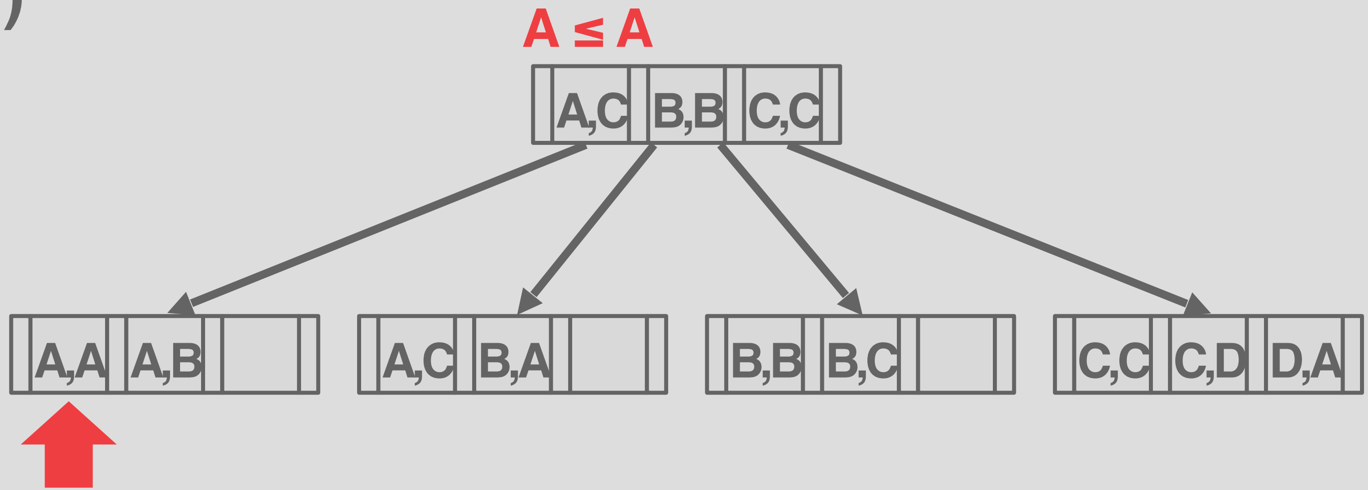
Find Key=(A,B)



SELECTION CONDITIONS

Find Key=(A,B)

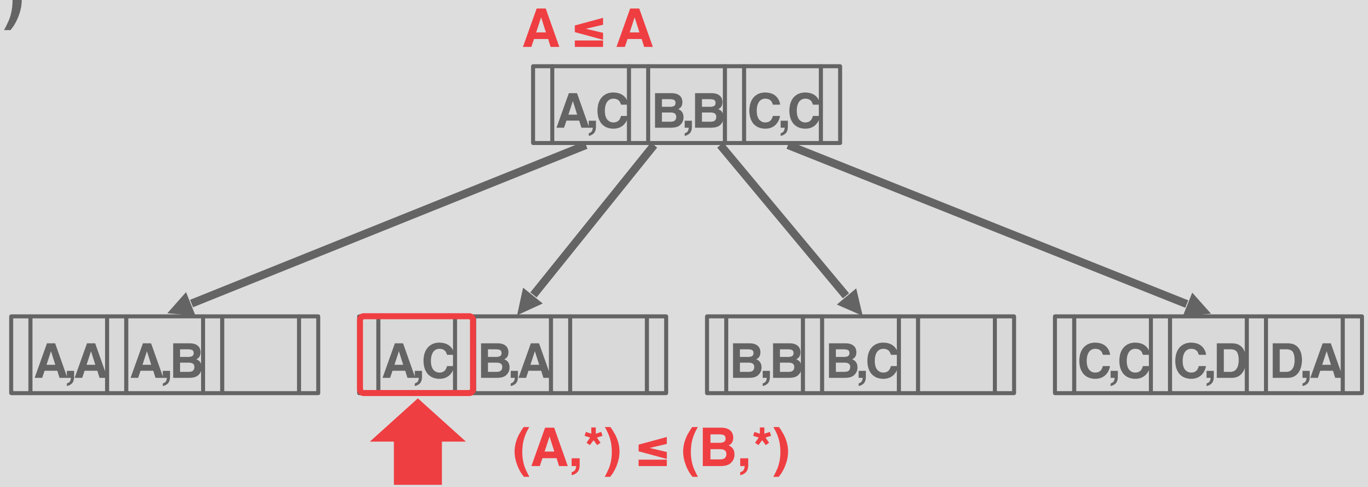
Find Key=(A,*)



SELECTION CONDITIONS

Find Key=(A,B)

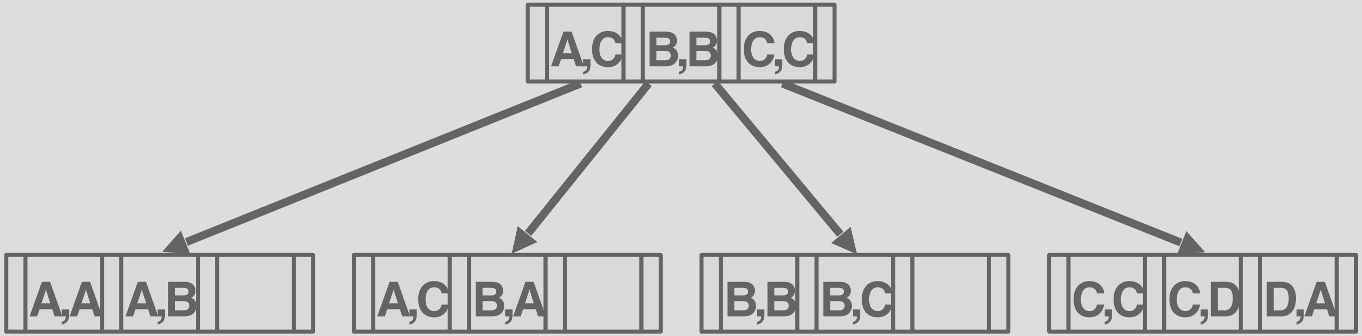
Find Key=(A,*)



SELECTION CONDITIONS

Find Key=(A,B)

Find Key=(A,*)

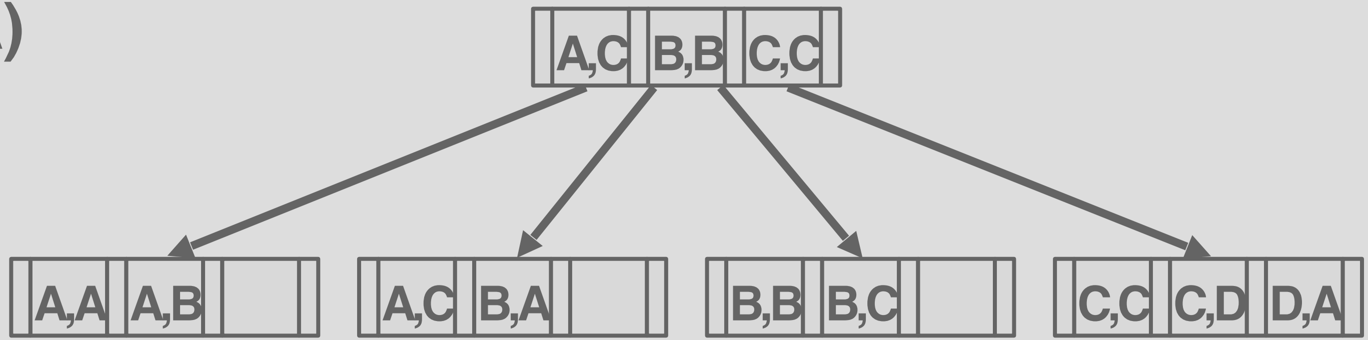


SELECTION CONDITIONS

Find Key=(A,B)

Find Key=(A,*)

Find Key=(*,A)

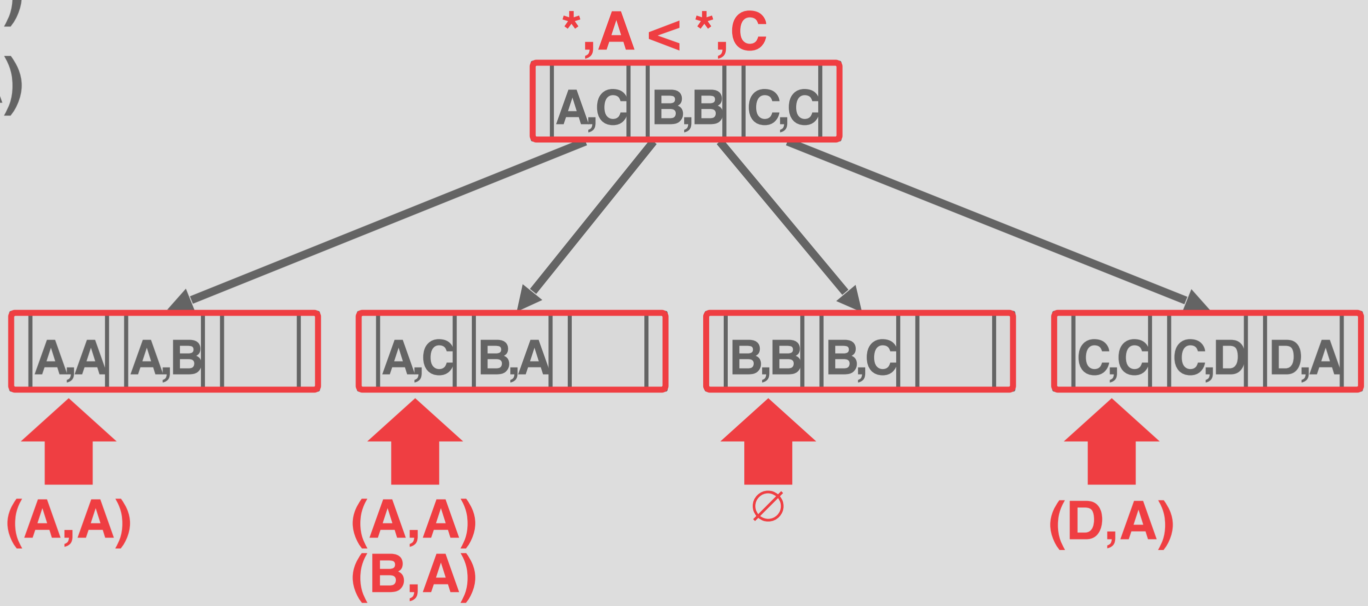


SELECTION CONDITIONS

Find Key=(A,B)

Find Key=(A,*)

Find Key=(*,A)

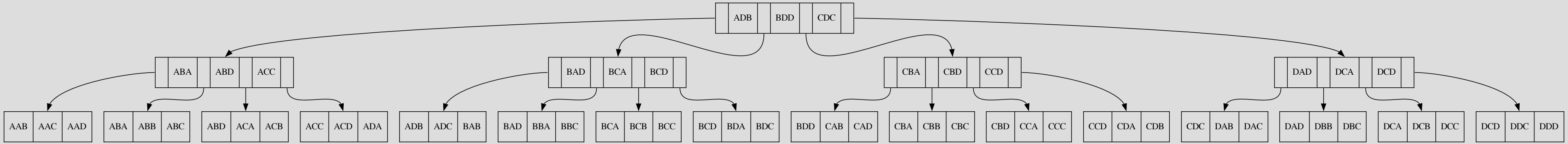


SELECTION CONDITIONS

Example: Index on **<col1,col2,col3>**

→ Column Values: **{A,B,C,D}**

→ Supported: **col2 = B**

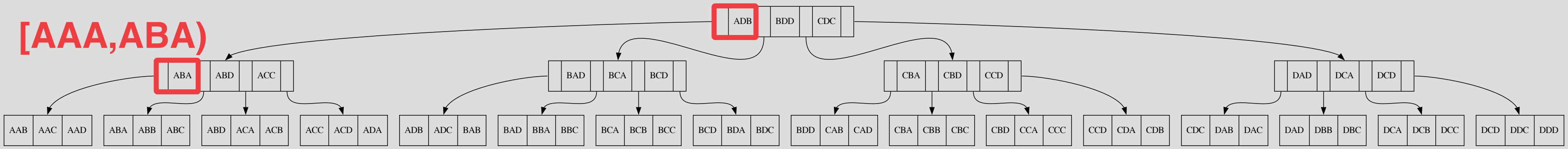


SELECTION CONDITIONS

Example: Index on **<col1,col2,col3>**

→ Column Values: **{A,B,C,D}**

→ Supported: **col2 = B**

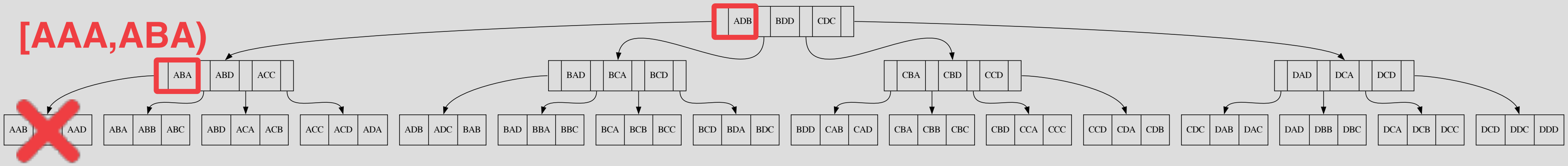


SELECTION CONDITIONS

Example: Index on **<col1,col2,col3>**

→ Column Values: **{A,B,C,D}**

→ Supported: **col2 = B**

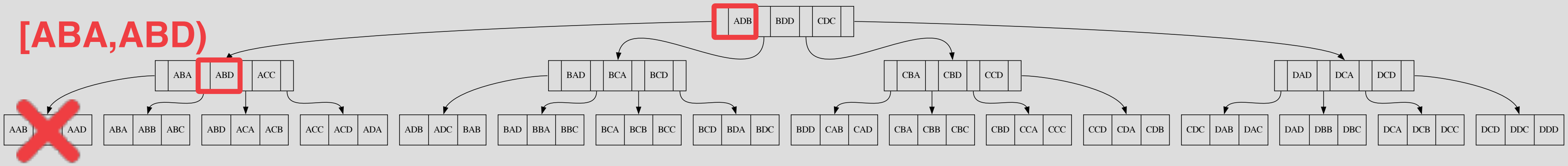


SELECTION CONDITIONS

Example: Index on **<col1,col2,col3>**

→ Column Values: **{A,B,C,D}**

→ Supported: **col2 = B**

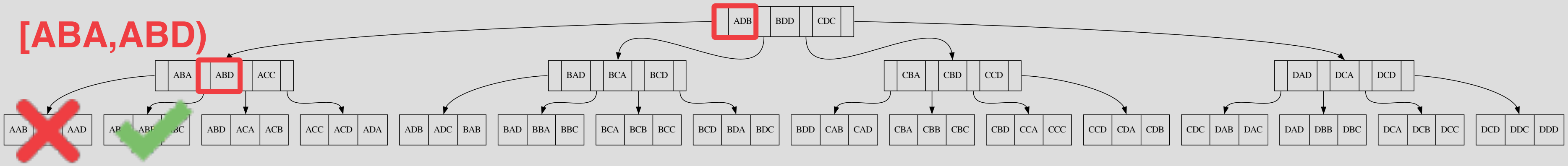


SELECTION CONDITIONS

Example: Index on **<col1,col2,col3>**

→ Column Values: **{A,B,C,D}**

→ Supported: **col2 = B**

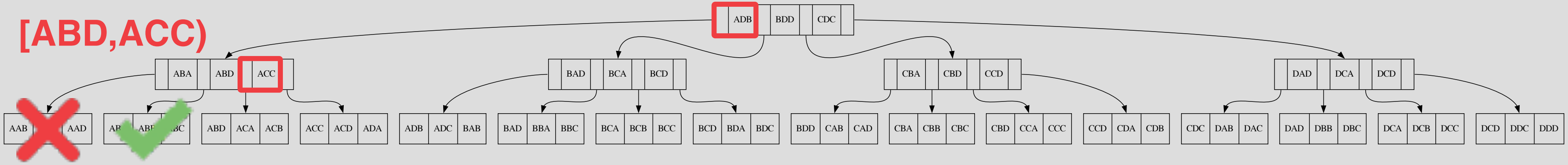


SELECTION CONDITIONS

Example: Index on **<col1,col2,col3>**

→ Column Values: **{A,B,C,D}**

→ Supported: **col2 = B**

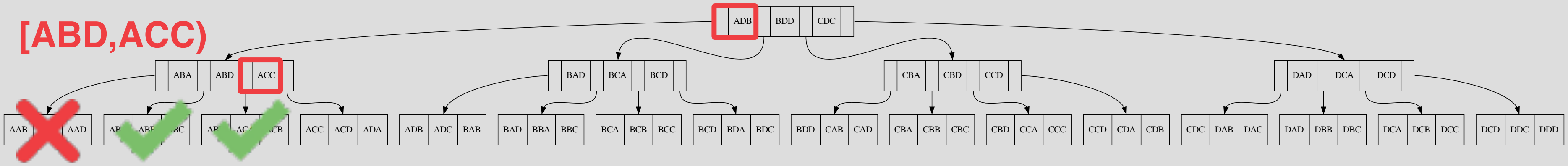


SELECTION CONDITIONS

Example: Index on **<col1,col2,col3>**

→ Column Values: **{A,B,C,D}**

→ Supported: **col2 = B**

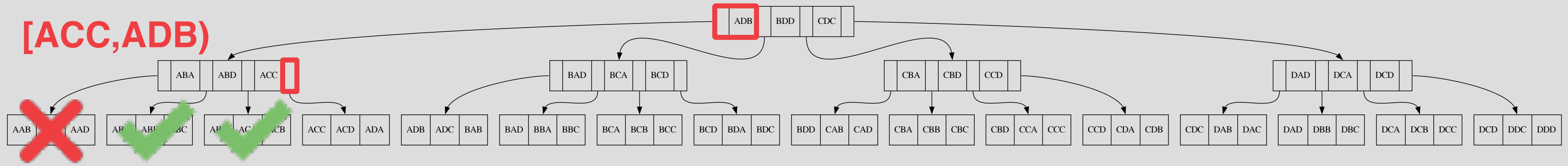


SELECTION CONDITIONS

Example: Index on **<col1,col2,col3>**

→ Column Values: **{A,B,C,D}**

→ Supported: **col2 = B**

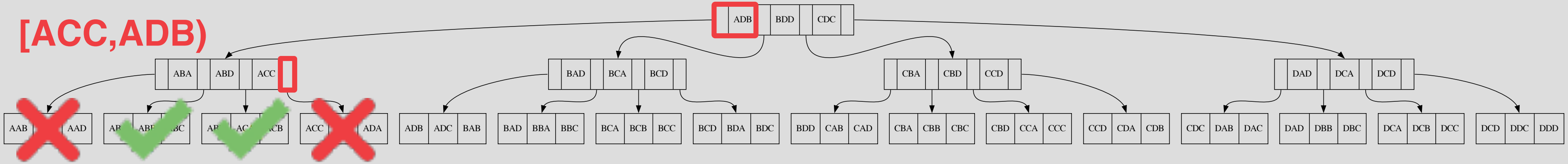


SELECTION CONDITIONS

Example: Index on **<col1,col2,col3>**

→ Column Values: **{A,B,C,D}**

→ Supported: **col2 = B**

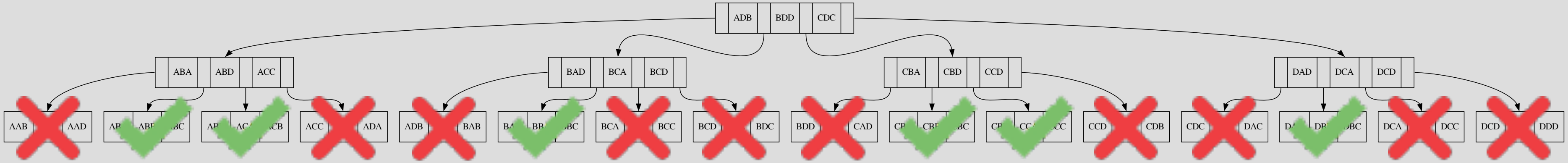


SELECTION CONDITIONS

Example: Index on **<col1,col2,col3>**

→ Column Values: **{A,B,C,D}**

→ Supported: **col2 = B**



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.

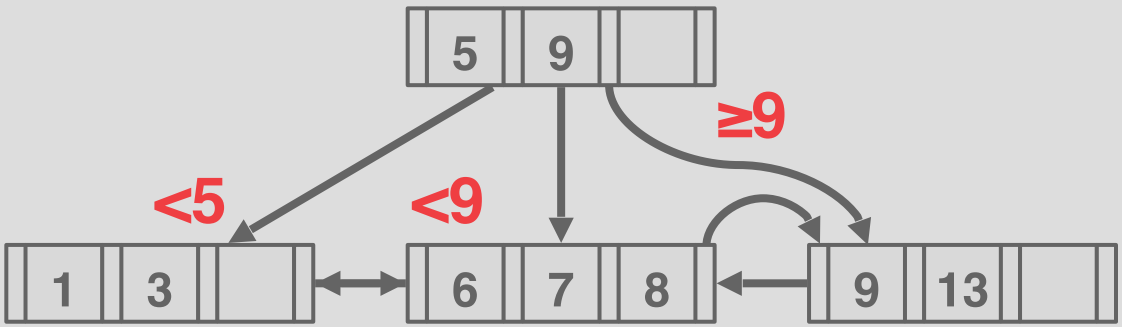
→ The DBMS can still use partial keys to find tuples.

Approach #2: Overflow Leaf Nodes

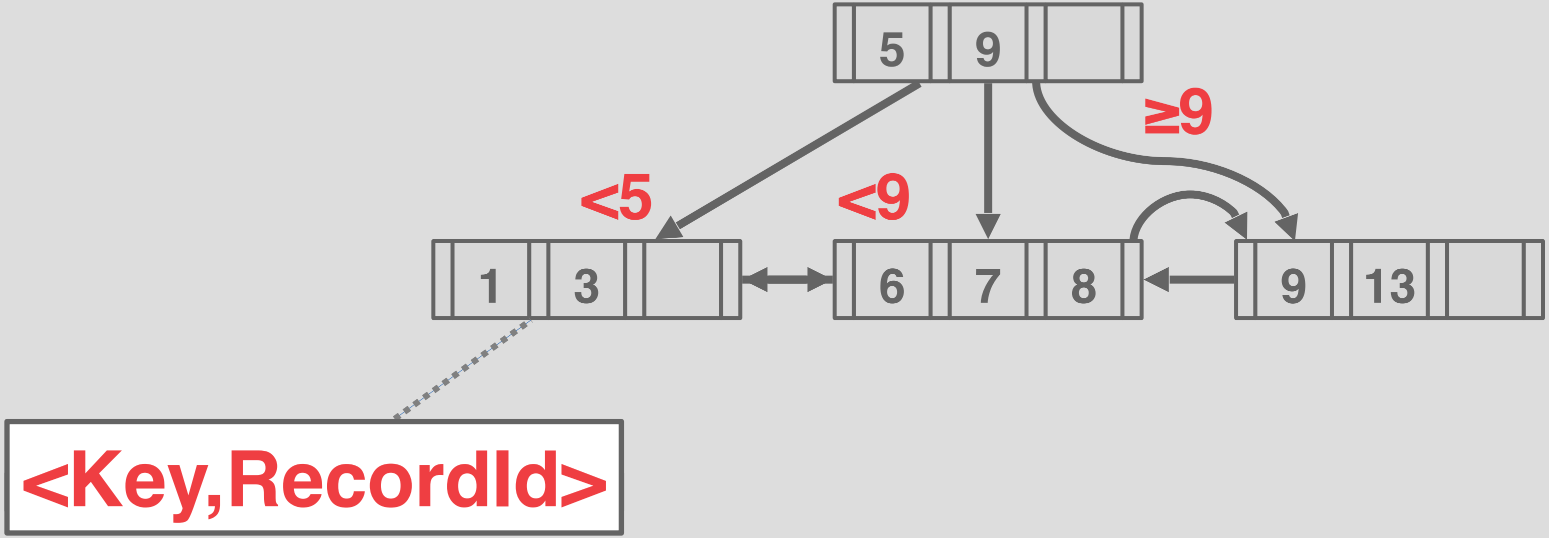
→ Allow leaf nodes to spill into overflow nodes that contain the duplicate keys.

→ This is more complex to maintain and modify.

B+TREE – APPEND RECORD ID

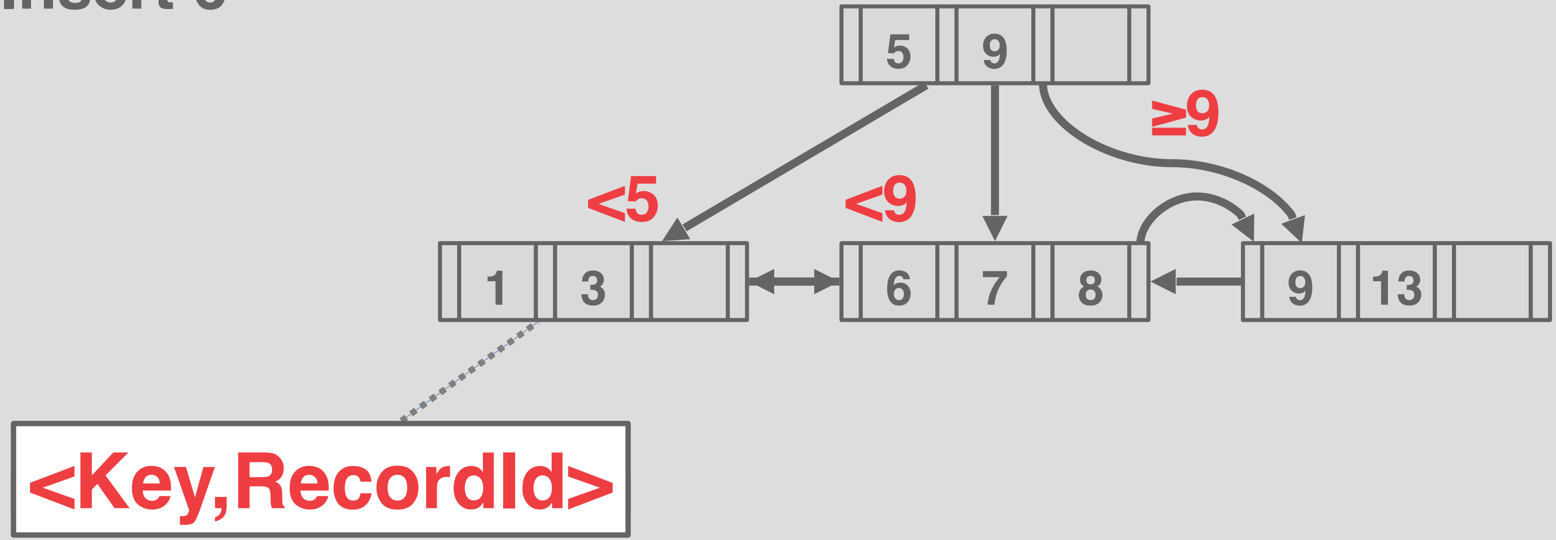


B+TREE – APPEND RECORD ID



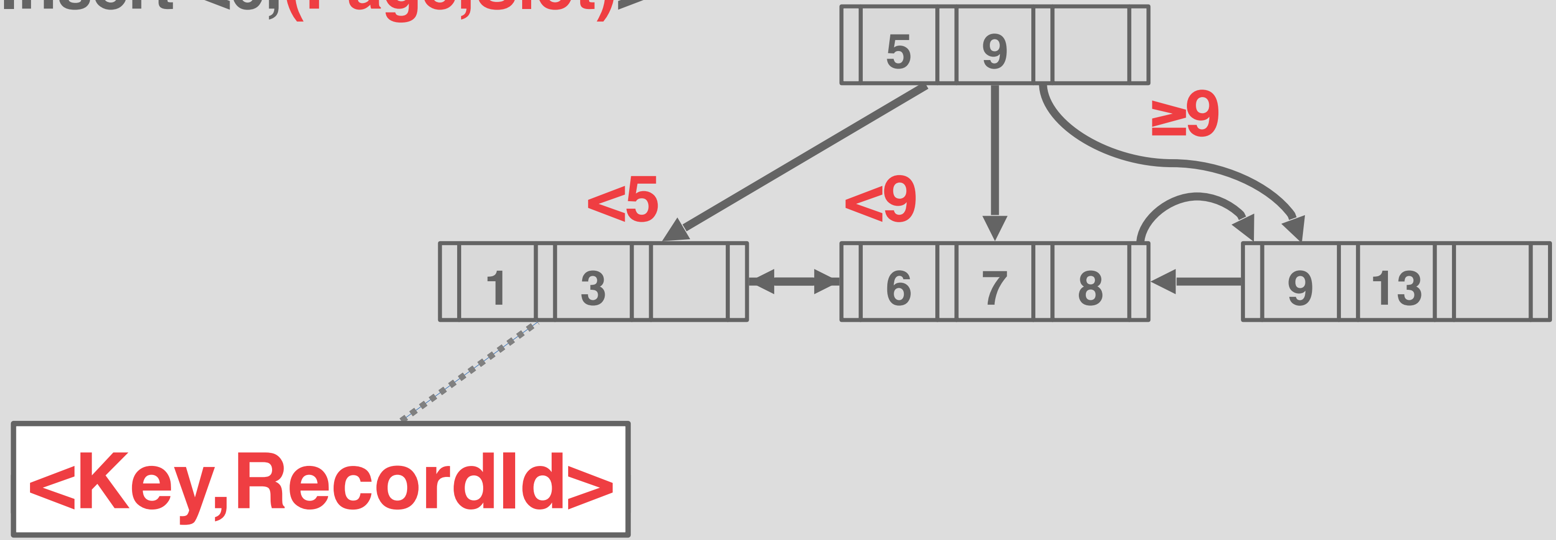
B+TREE – APPEND RECORD ID

Insert 6



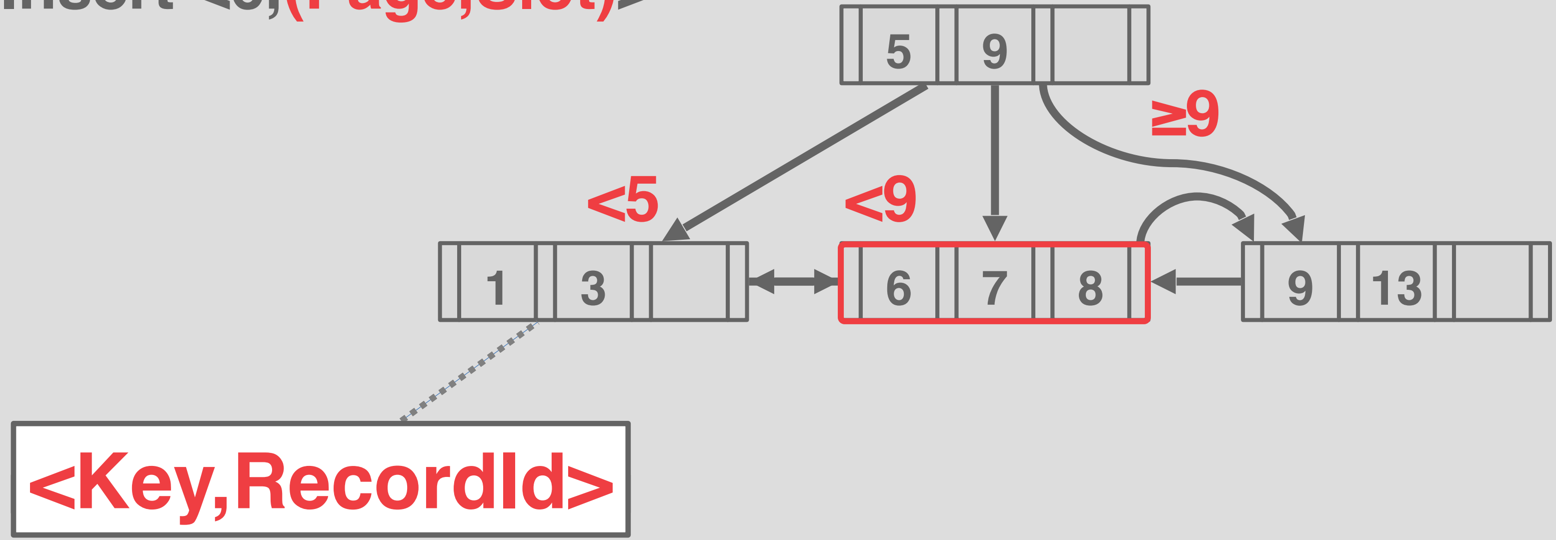
B+TREE – APPEND RECORD ID

Insert <6,(Page,Slot)>



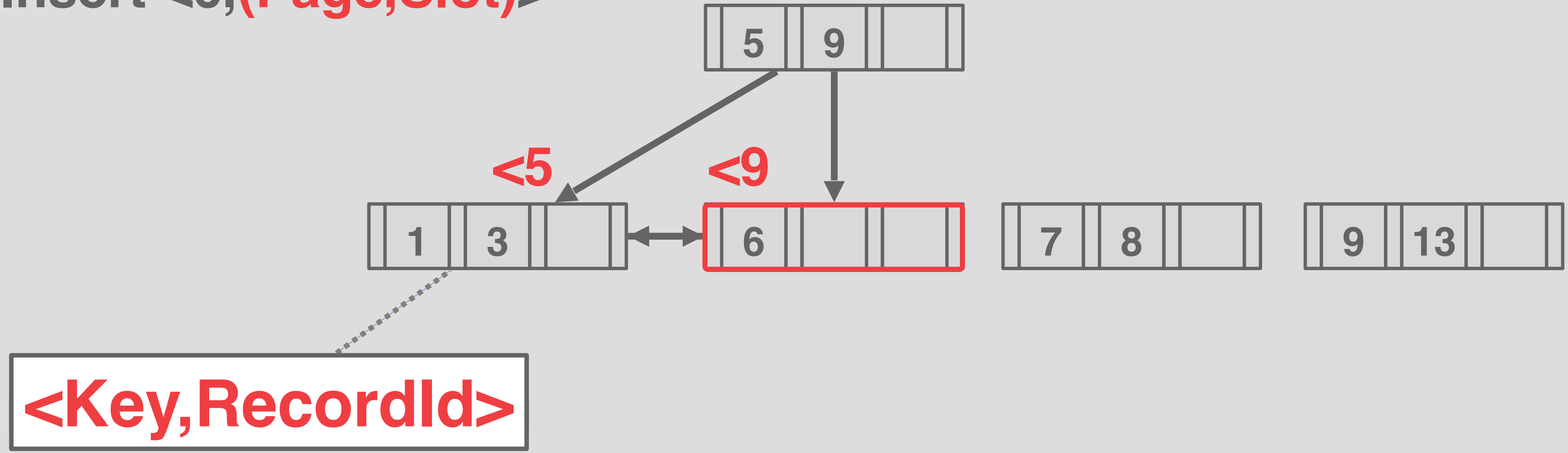
B+TREE – APPEND RECORD ID

Insert <6,(Page,Slot)>



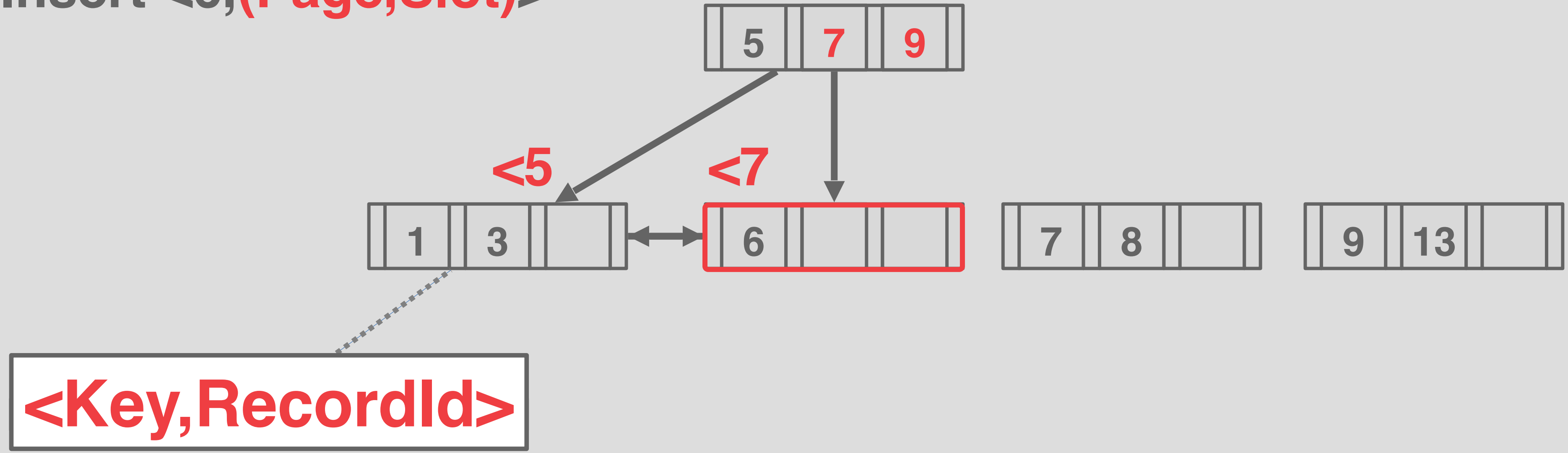
B+TREE – APPEND RECORD ID

Insert <6,(Page,Slot)>



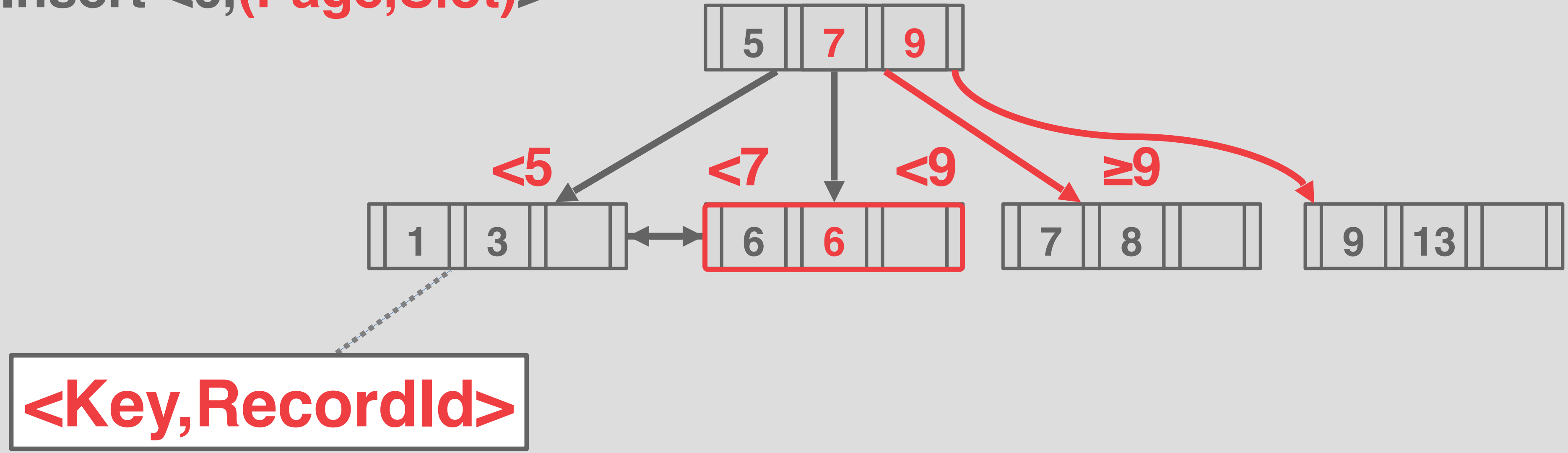
B+TREE – APPEND RECORD ID

Insert <6,(Page,Slot)>



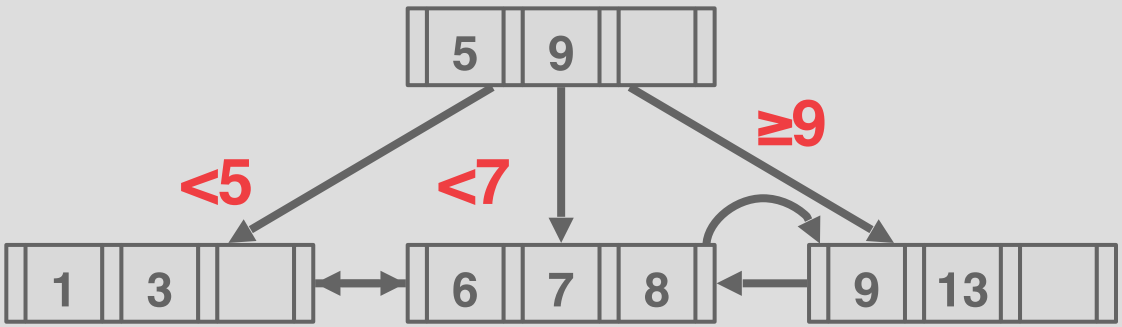
B+TREE – APPEND RECORD ID

Insert <6,(Page,Slot)>



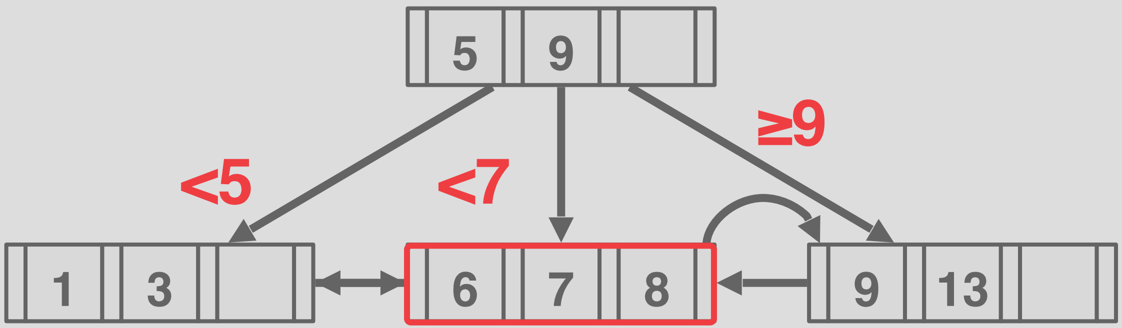
B+TREE – OVERFLOW LEAF NODES

Insert 6



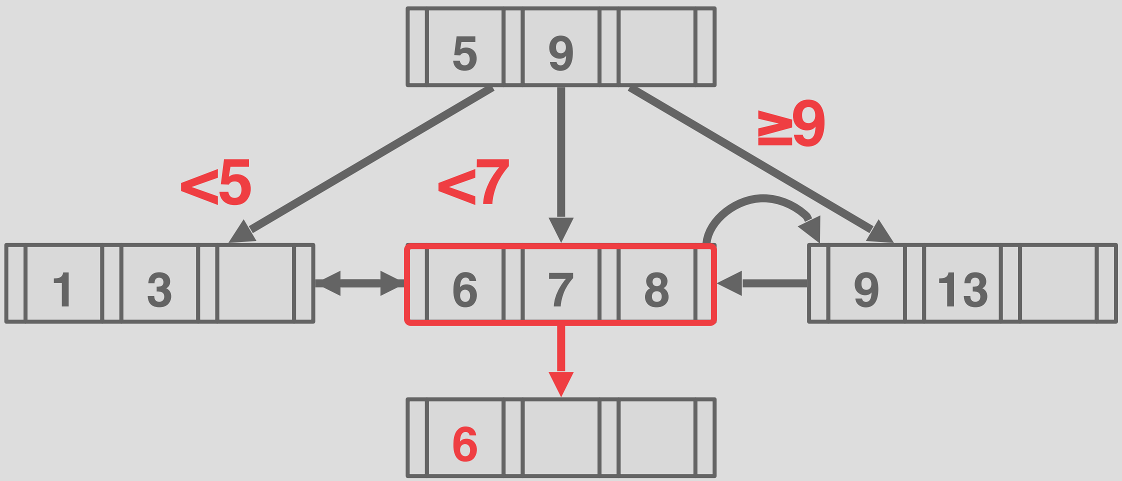
B+TREE – OVERFLOW LEAF NODES

Insert 6



B+TREE – OVERFLOW LEAF NODES

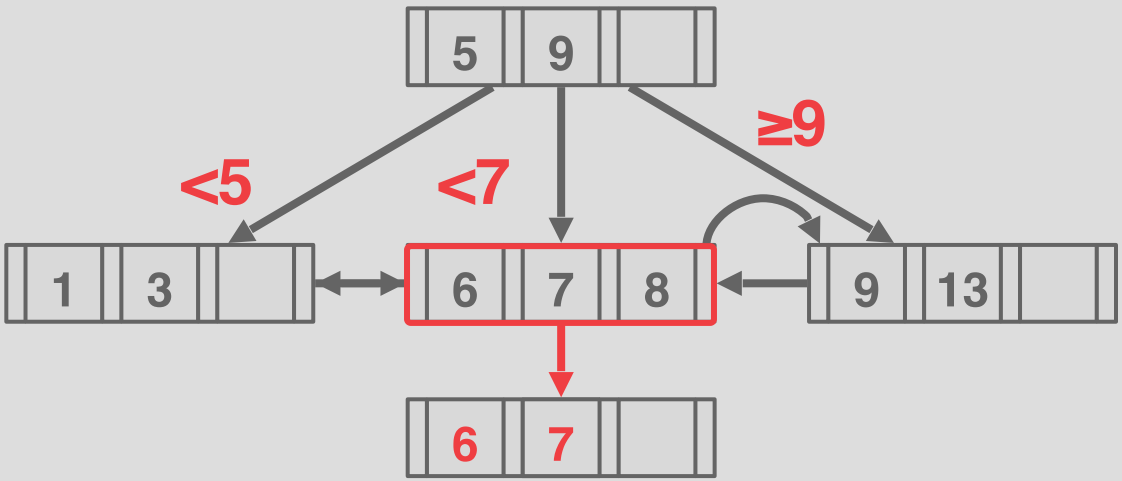
Insert 6



B+TREE – OVERFLOW LEAF NODES

Insert 6

Insert 7

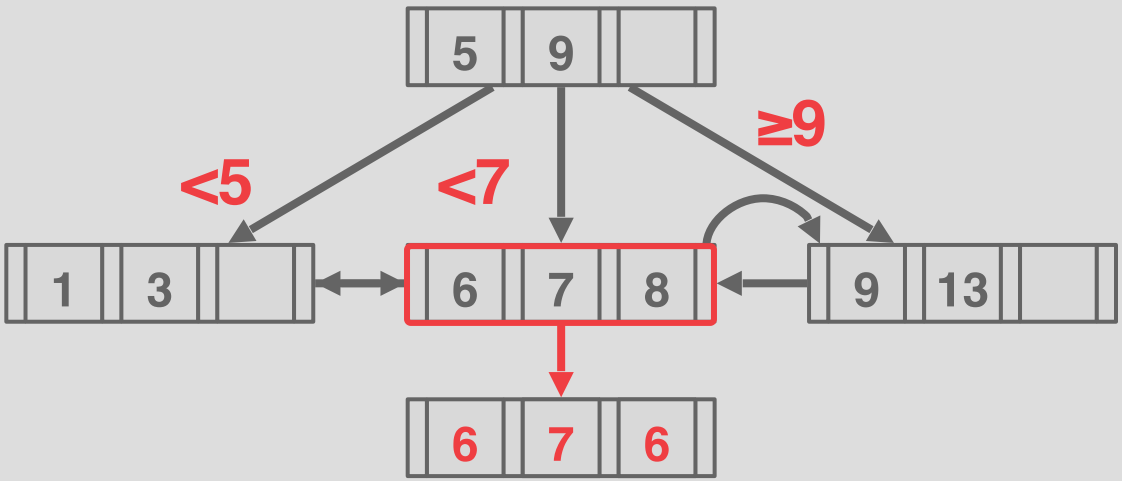


B+TREE – OVERFLOW LEAF NODES

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.

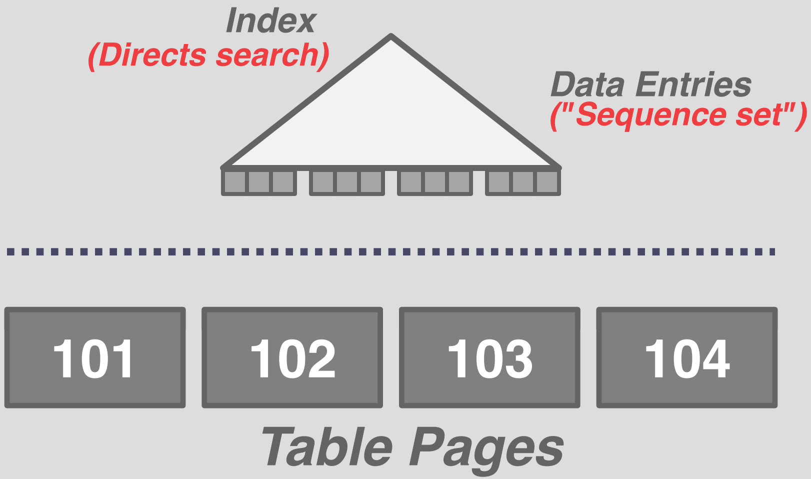
Some DBMSs always use a clustered index.
→ If a table does not contain a primary key, the DBMS will automatically make a hidden primary key.

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.

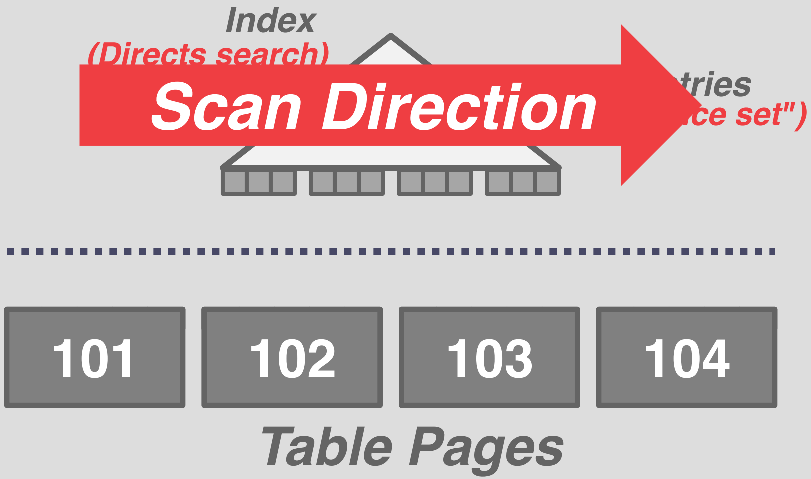
This will always be better than sorting data for each query.



CLUSTERED B+TREE

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

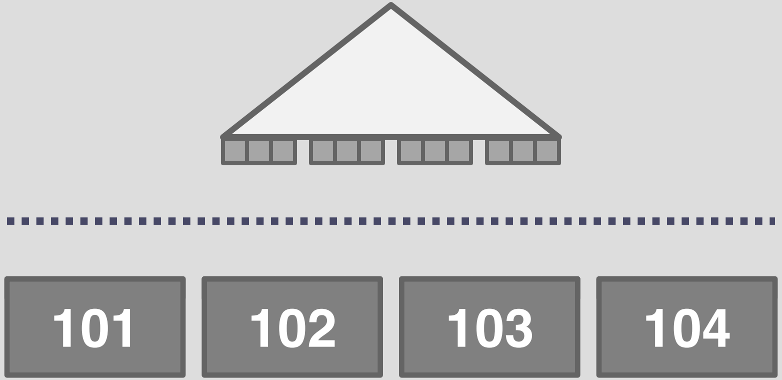
This will always be better than sorting data for each query.



INDEX SCAN PAGE SORTING

Retrieving tuples in the order they appear in a non-clustered index is inefficient due to redundant reads.

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

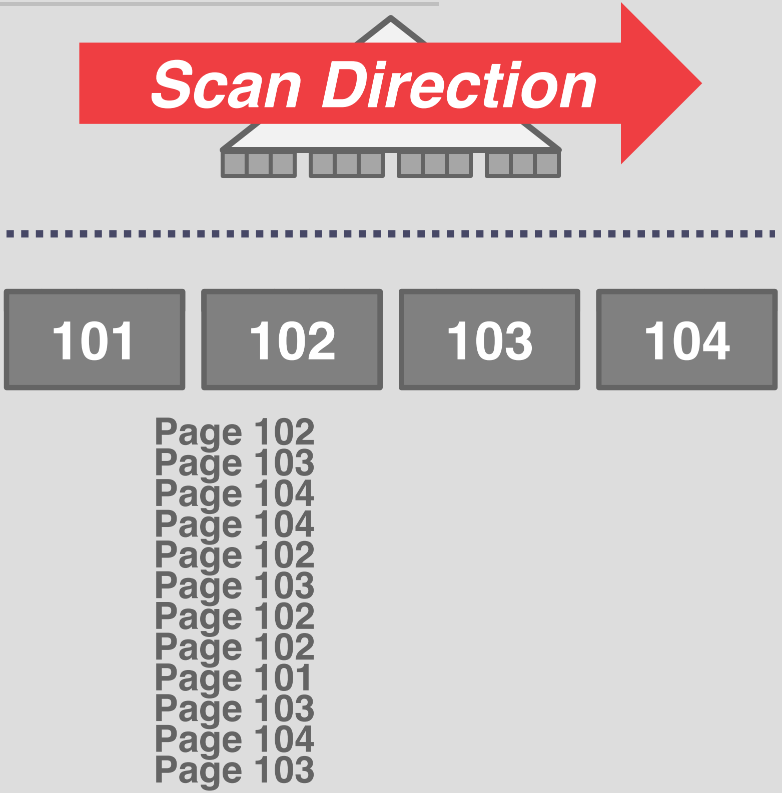
Retrieving tuples in the order they appear in a non-clustered index is inefficient due to redundant reads.

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 is inefficient due to redundant reads.

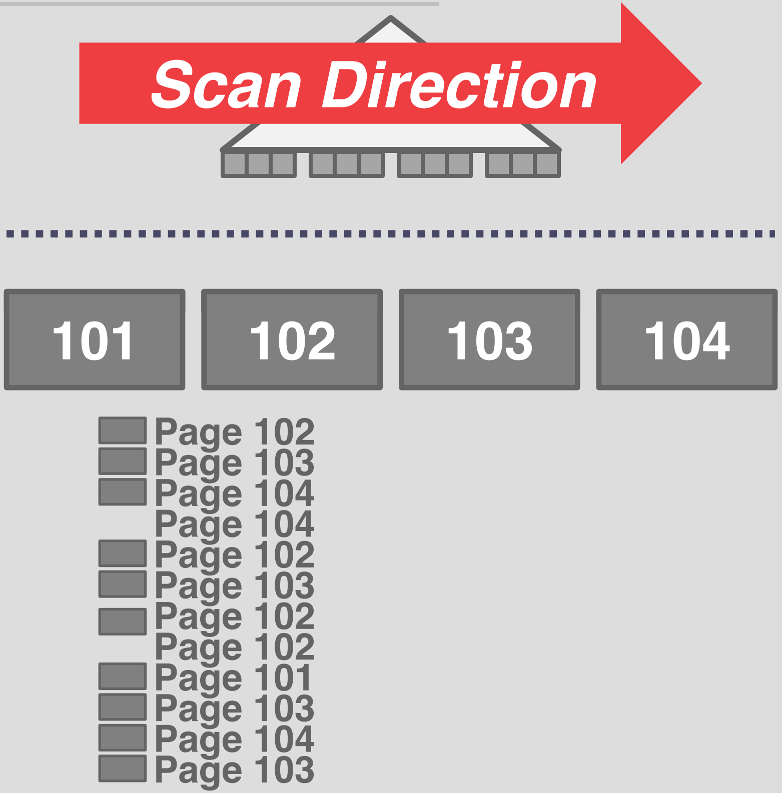
The DBMS can first figure out all the tuples that it needs and then sort them based on their Page ID.



INDEX SCAN PAGE SORTING

Retrieving tuples in the order they appear in a non-clustered index is inefficient due to redundant reads.

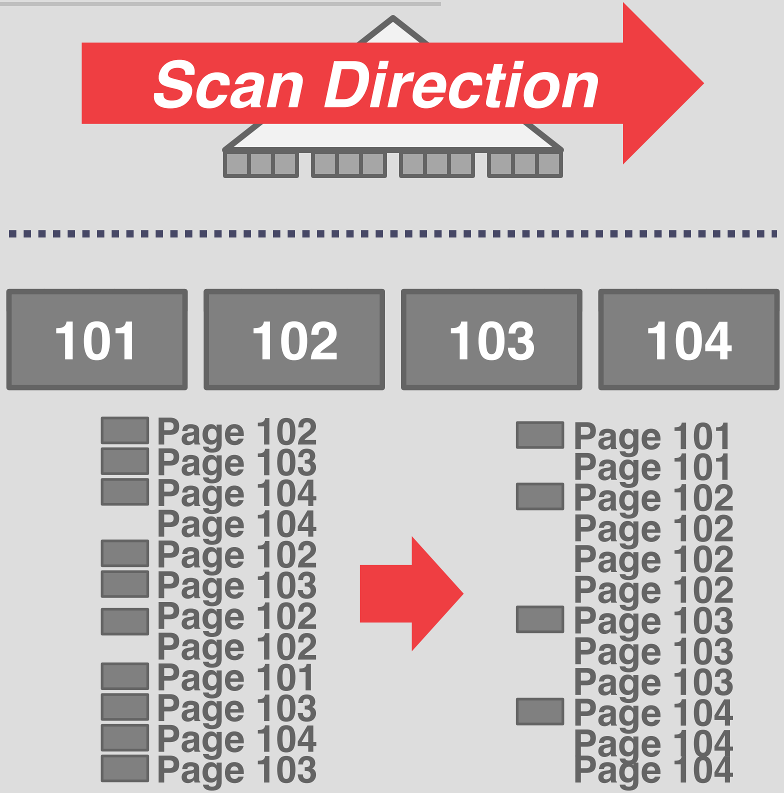
The DBMS can first figure out all the tuples that it needs and then sort them based on their Page ID.



INDEX SCAN PAGE SORTING

Retrieving tuples in the order they appear in a non-clustered index is inefficient due to redundant reads.

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



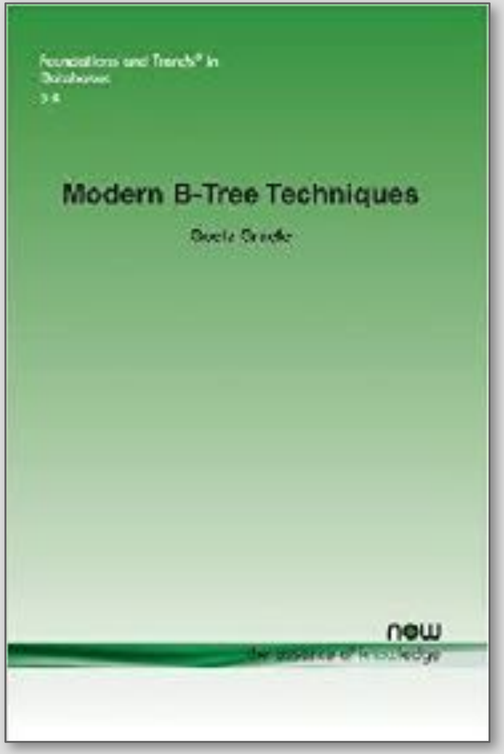
B+TREE DESIGN CHOICES

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.
→ Use SIMD to vectorize comparisons.

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.
→ Use SIMD to vectorize comparisons.

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.
→ Use SIMD to vectorize comparisons.

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.
→ Use SIMD to vectorize comparisons.

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

4	5	6	7	8	9	10

Approach #1: Linear

→ Scan node keys from beginning to end.
→ Use SIMD to vectorize comparisons.

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.
→ Use SIMD to vectorize comparisons.

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.

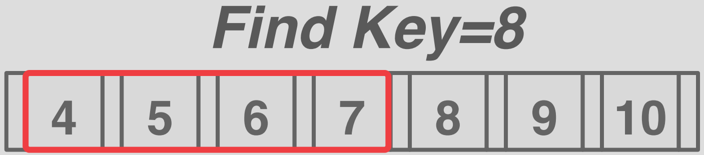
Find Key=8

INTRA-NODE SEARCH

Approach #1: Linear
→ Scan node keys from beginning to end.
→ Use SIMD to vectorize comparisons.

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.



`_mm_cmpeq_epi32_mask(a, b)`

INTRA-NODE SEARCH

Approach #1: Linear

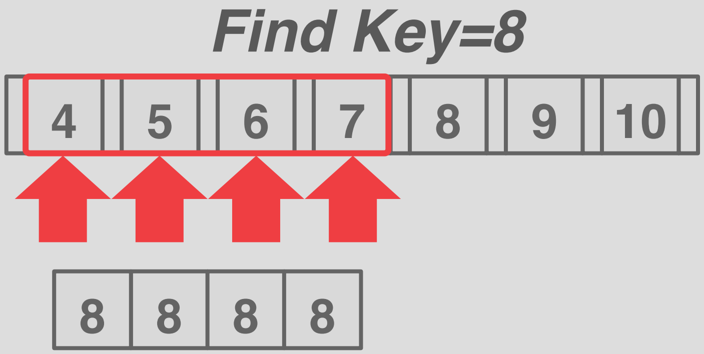
→ Scan node keys from beginning to end.
→ Use SIMD to vectorize comparisons.

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.



_mm_cmpeq_epi32_mask(a, b)

INTRA-NODE SEARCH

Approach #1: Linear
→ Scan node keys from beginning to end.
→ Use SIMD to vectorize comparisons.

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.

Find Key=8


```
_mm_cmpeq_epi32_mask(a, b)
```

INTRA-NODE SEARCH

Approach #1: Linear

→ Scan node keys from beginning to end.
→ Use SIMD to vectorize comparisons.

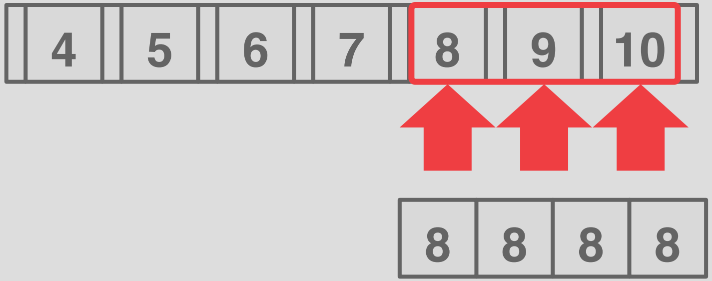
Approach #2: Binary

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

Approach #3: Interpolation

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

Find Key=8



_mm_cmpeq_epi32_mask(a, b)

INTRA-NODE SEARCH

Find Key=8

Approach #1: Linear

→ Scan node keys from beginning to end.
→ Use SIMD to vectorize comparisons.

Approach #2: Binary

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

Approach #3: Interpolation

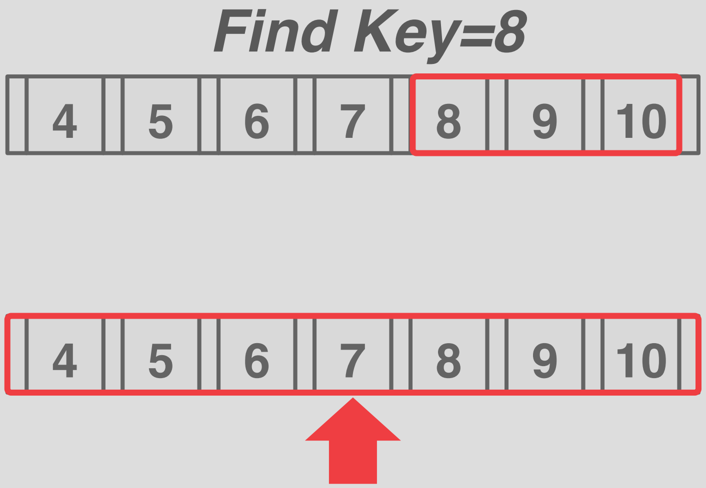
→ Approximate location of desired key based on known distribution of keys.

INTRA-NODE SEARCH

Approach #1: Linear
→ Scan node keys from beginning to end.
→ Use SIMD to vectorize comparisons.

Approach #2: Binary
→ Jump to middle key, pivot left/right depending on comparison.

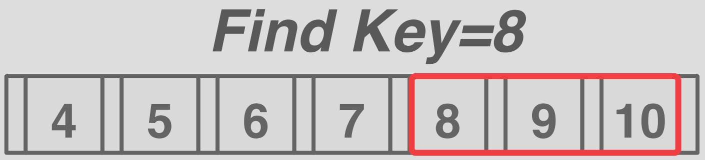
Approach #3: Interpolation
→ Approximate location of desired key based on known distribution of keys.



INTRA-NODE SEARCH

Approach #1: Linear

→ Scan node keys from beginning to end.
→ Use SIMD to vectorize comparisons.



Approach #2: Binary

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

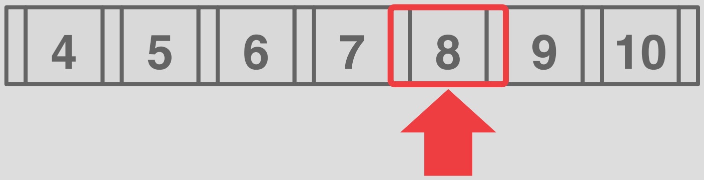
Approach #3: Interpolation

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

INTRA-NODE SEARCH

Approach #1: Linear

→ Scan node keys from beginning to end.
→ Use SIMD to vectorize comparisons.



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.

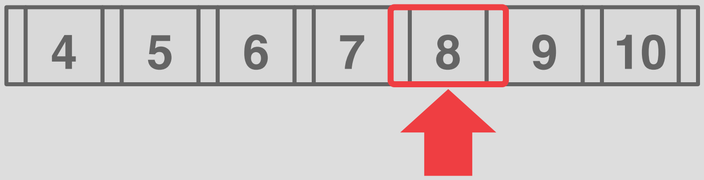
INTRA-NODE SEARCH

Approach #1: Linear

→ Scan node keys from beginning to end.
→ Use SIMD to vectorize comparisons.

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.
→ Use SIMD to vectorize comparisons.

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

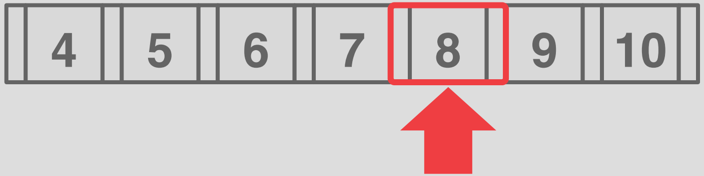
Find Key=8

Approach #1: Linear

→ Scan node keys from beginning to end.
→ Use SIMD to vectorize comparisons.

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.

*Offset: (8-4)*7/(10-4)=4*

OPTIMIZATIONS

Prefix Compression

Deduplication

Suffix Truncation

Pointer Swizzling

Bulk Insert

Buffer Updates

Many more…

PREFIX COMPRESSION

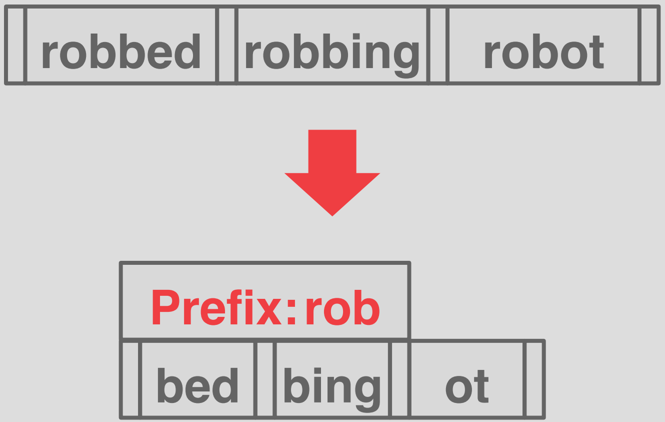
Sorted keys in the same leaf node are likely to have the same prefix.

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

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.

K1	V1	K1	V2	K1	V3	K2	V4

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).

DEDUPLICATION

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

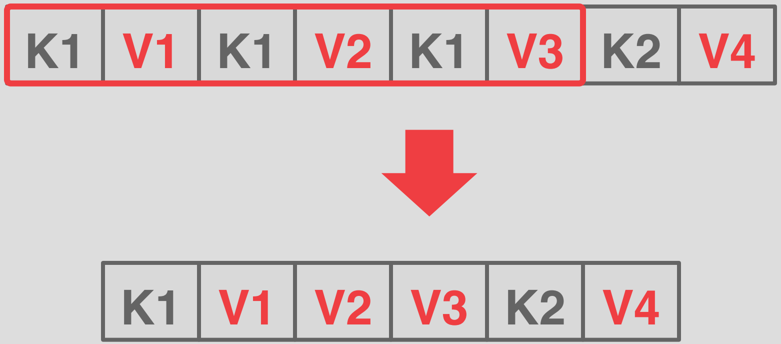
The leaf node can store the key once and then maintain a list of tuples with that key (similar to what we discussed for hash tables).

K1	V1	K1	V2	K1	V3	K2	V4

DEDUPLICATION

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

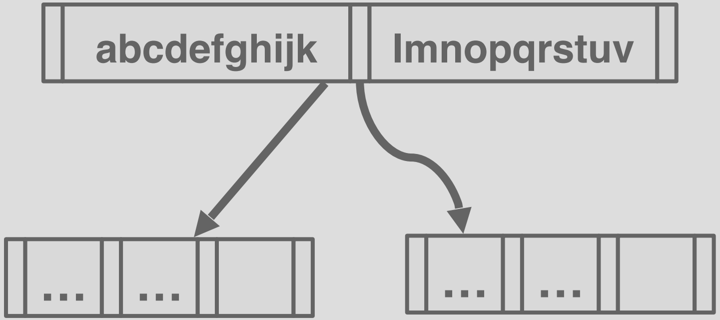
The leaf node can store the key once and then maintain a list of tuples with that key (similar to what we discussed for hash tables).



SUFFIX TRUNCATION

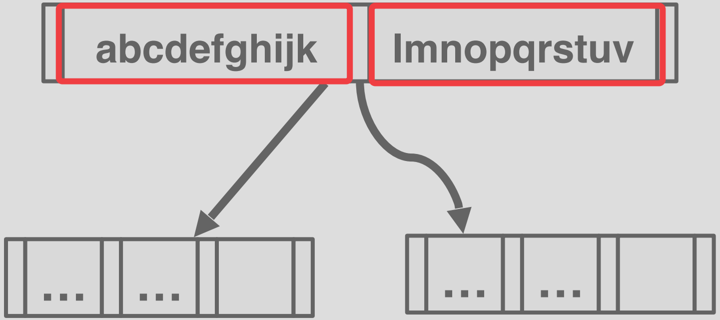
The keys in the inner nodes are only used to "direct traffic".
→ We don't need the entire key.

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.

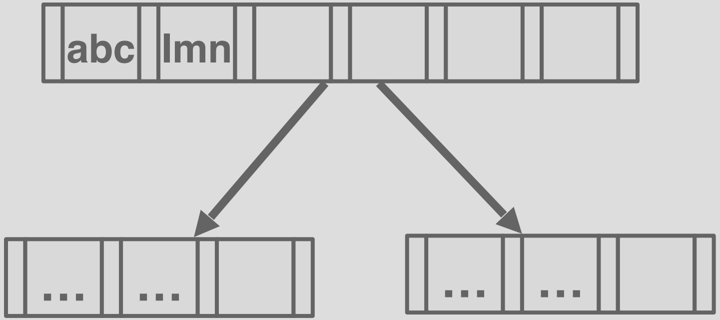


Store a minimum prefix that is needed to correctly route probes into the index.

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.



POINTER SWIZZLING

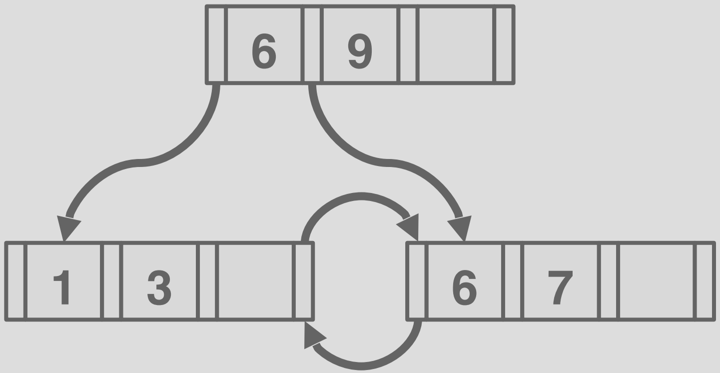
Nodes use page ids to reference other nodes in the index. The DBMS must get the memory location from the page table during traversal.

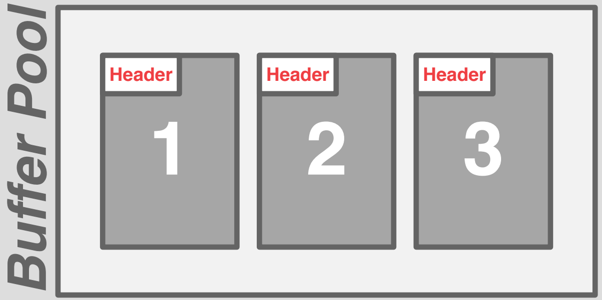
If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids. This avoids address lookups from the page table.

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.

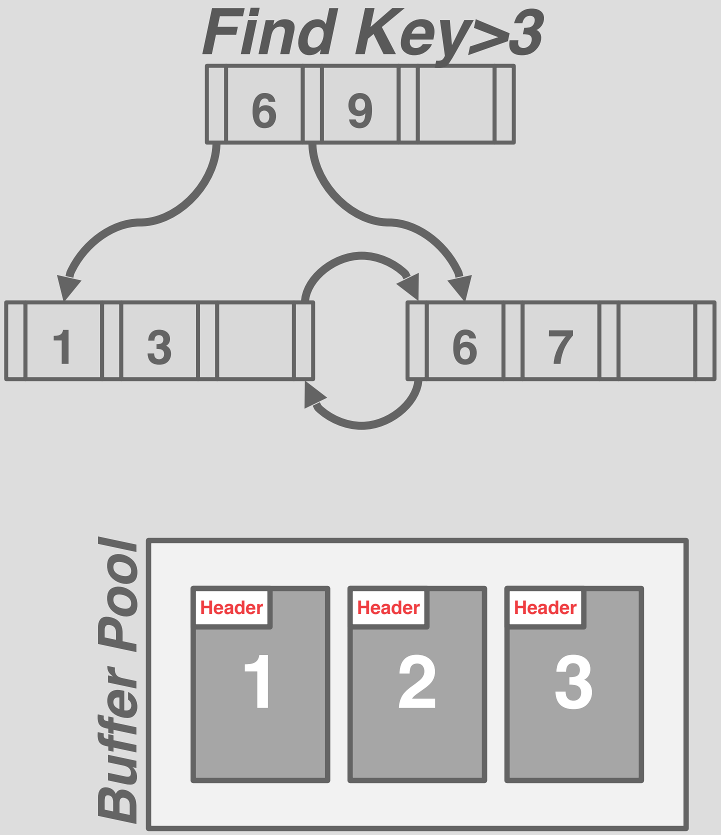




POINTER SWIZZLING

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

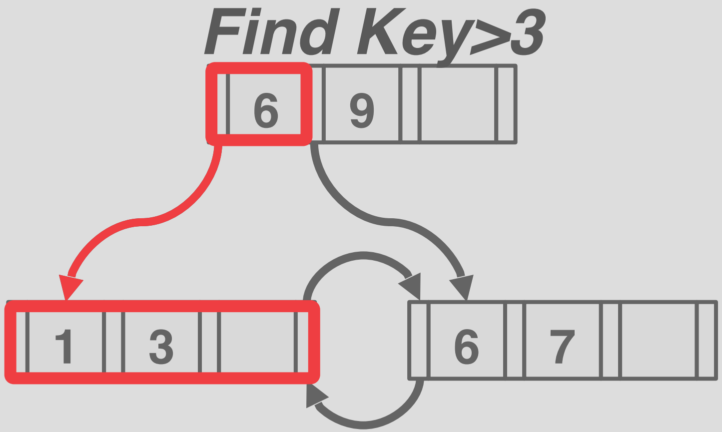
If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids. This avoids address lookups from the page table.

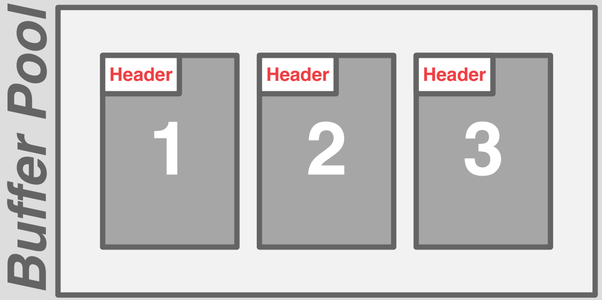


POINTER SWIZZLING

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.

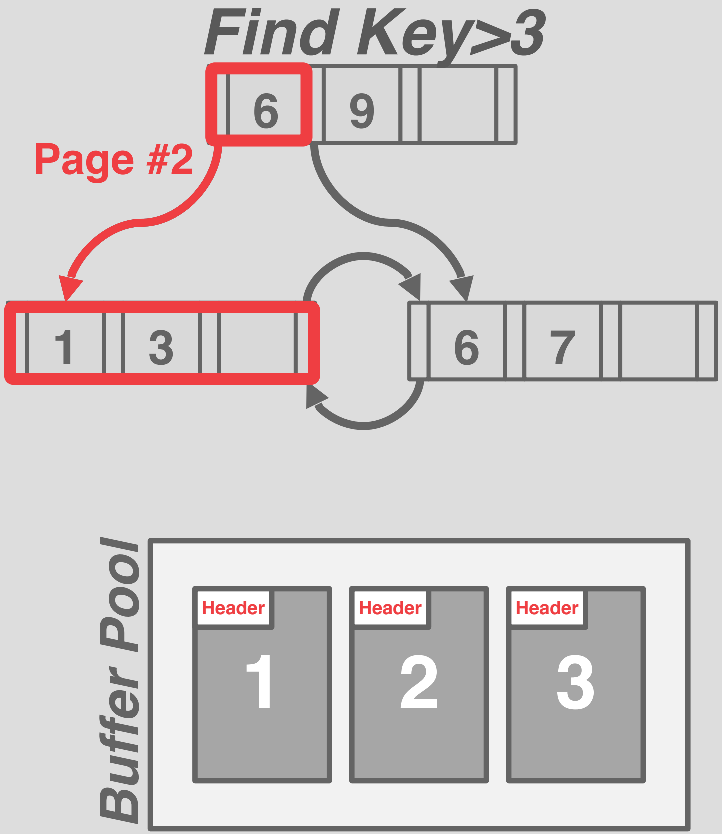




POINTER SWIZZLING

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

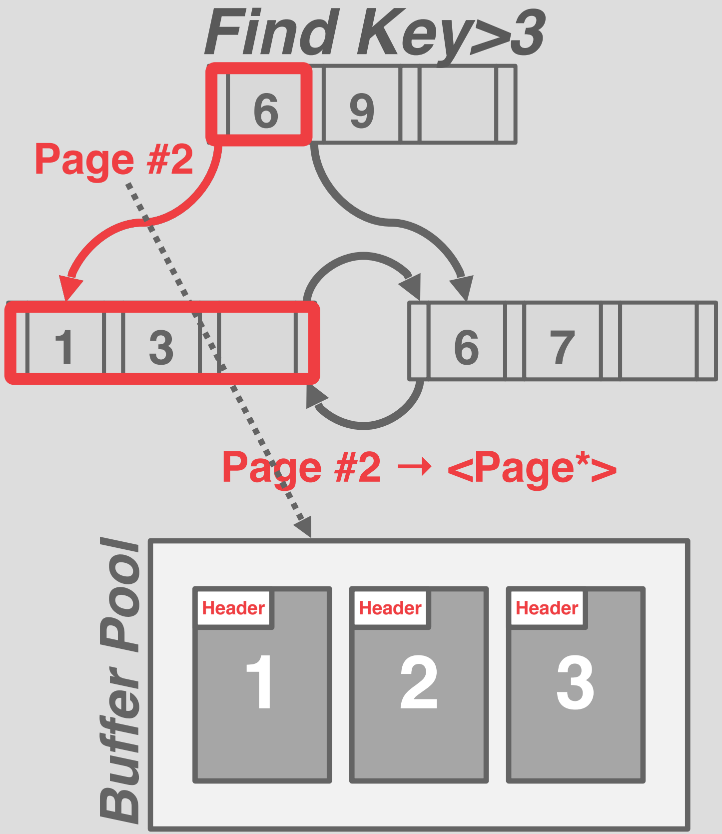
If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids. This avoids address lookups from the page table.



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.

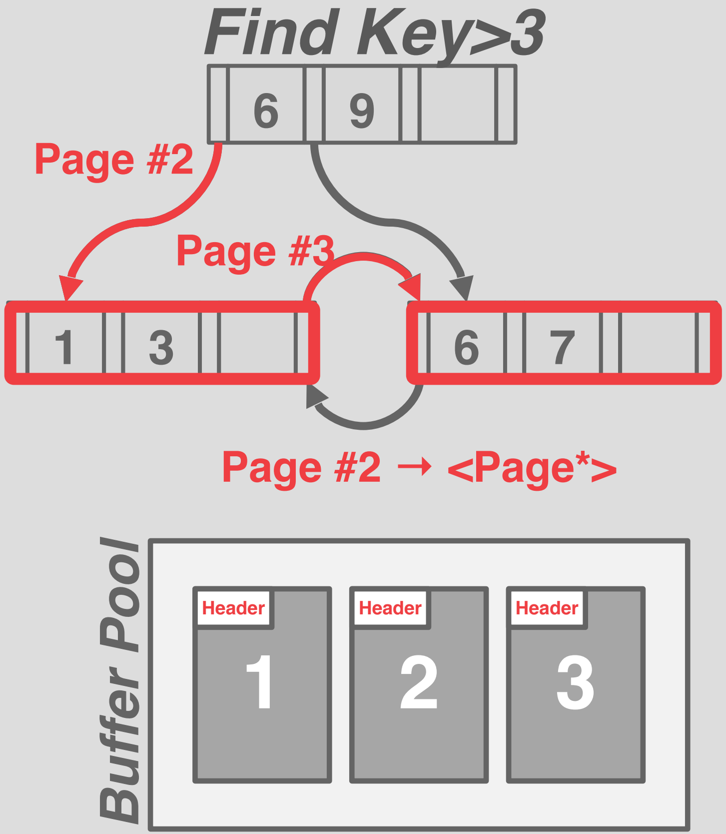
If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids. This avoids address lookups from the page table.



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.

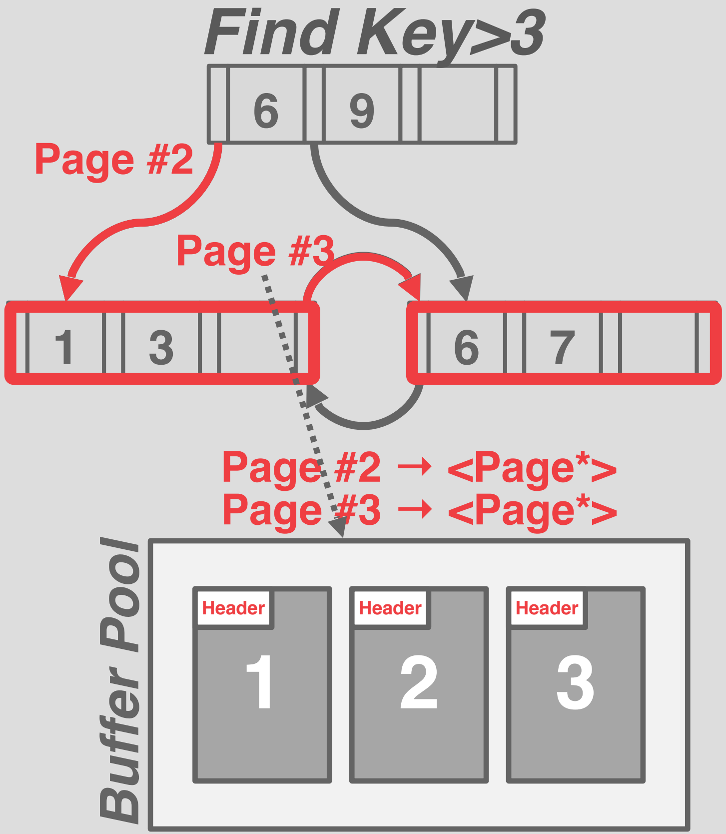
If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids. This avoids address lookups from the page table.



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.

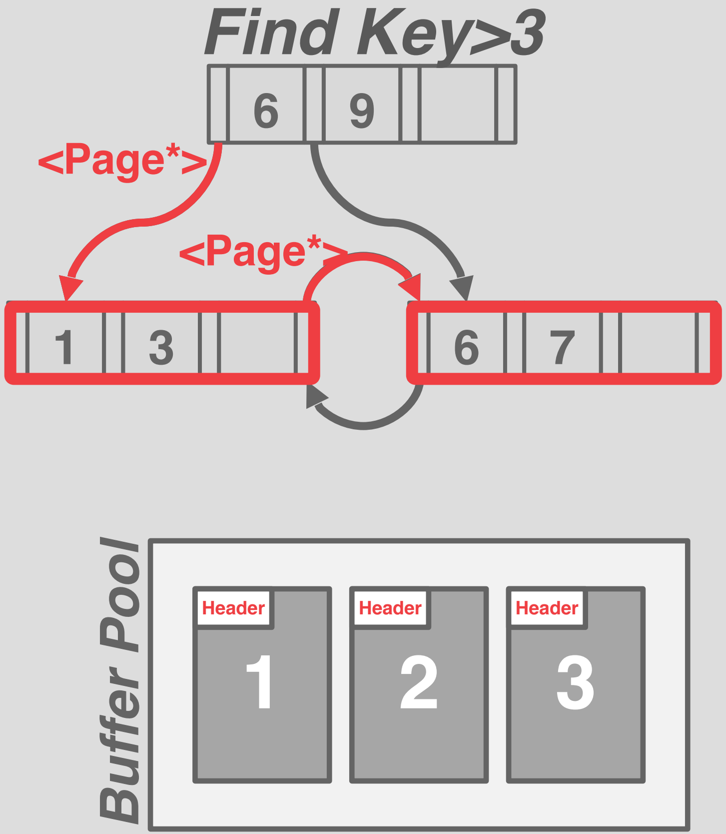
If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids. This avoids address lookups from the page table.



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.



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.

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

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

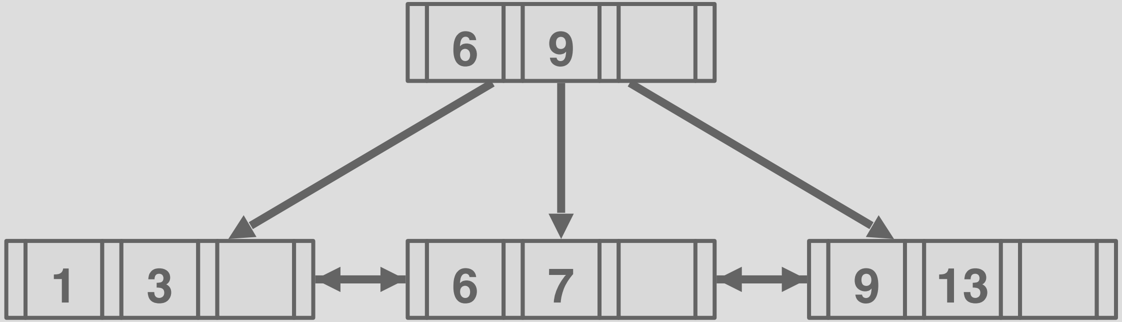
Sorted Keys: 1, 3, 6, 7, 9, 13

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 B+Tree is (almost) always a good choice for an index in your DBMS.

NEXT CLASS

Index Concurrency Control