Carnegie Mellon University

07 Tree Indexes – Part I

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

AP
Andy Pavlo
Computer Science
Carnegie Mellon University

ADMINISTRIVIA

Project #1 is due Sunday Sept 27th

Homework #2 is due Sunday Oct 4th

UPCOMING DATABASE TALKS

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

Apache Arrow
→ Monday Oct 5th @ 5pm ET

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

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 a subset of those attributes.

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

Using B+Trees in a DBMS

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:

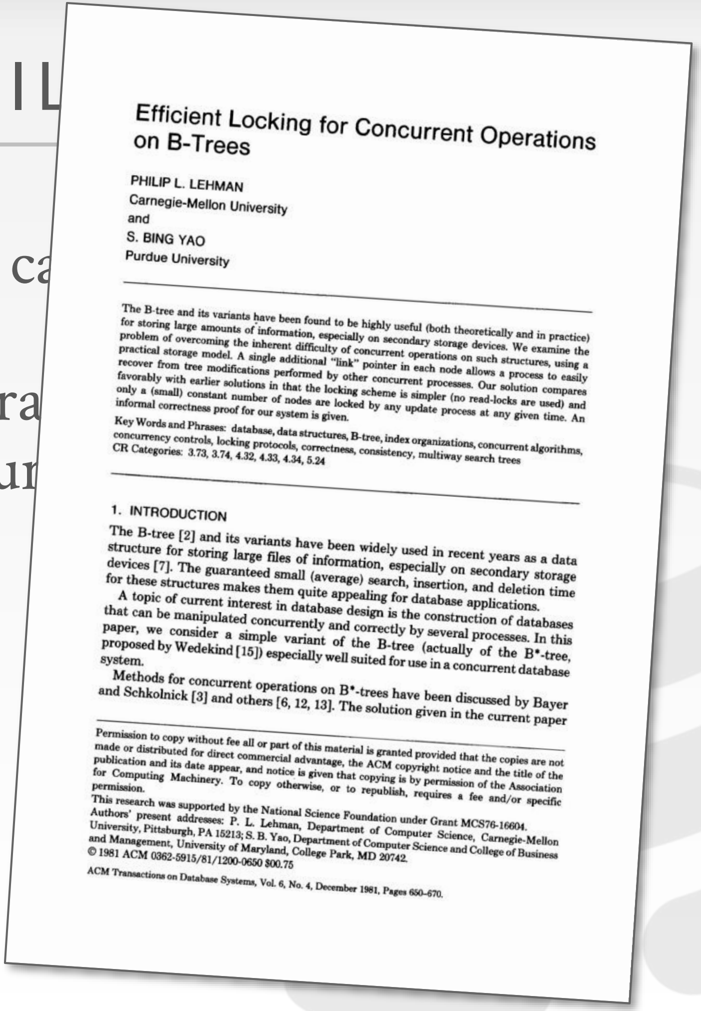
→ **B-Tree** (1971)
→ **B+Tree** (1973)
→ **B*Tree** (1977?)
→ **B^{link}-Tree** (1981)

B-TREE FAMIL

There is a specific data structure ca

People also use the term to genera
class of balanced tree data structur

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



Efficient Locking for Concurrent Operations on B-Trees

PHILIP L. LEHMAN
Carnegie-Mellon University
and
S. BING YAO
Purdue University

The B-tree and its variants have been found to be highly useful (both theoretically and in practice) for storing large amounts of information, especially on secondary storage devices. We examine the problem of overcoming the inherent difficulty of concurrent operations on such structures, using a practical storage model. A single additional "link" pointer in each node allows a process to easily recover from tree modifications performed by other concurrent processes. Our solution compares favorably with earlier solutions in that the locking scheme is simpler (no read-locks are used) and only a (small) constant number of nodes are locked by any update process at any given time. An informal correctness proof for our system is given.

Key Words and Phrases: database, data structures, B-tree, index organizations, concurrent algorithms, concurrency controls, locking protocols, correctness, consistency, multiway search trees
CR Categories: 3.73, 3.74, 4.32, 4.33, 4.34, 5.24

1. INTRODUCTION

The B-tree [2] and its variants have been widely used in recent years as a data structure for storing large files of information, especially on secondary storage devices [7]. The guaranteed small (average) search, insertion, and deletion time for these structures makes them quite appealing for database applications.

A topic of current interest in database design is the construction of databases that can be manipulated concurrently and correctly by several processes. In this paper, we consider a simple variant of the B-tree (actually of the B*-tree, proposed by Wedekind [15]) especially well suited for use in a concurrent database system.

Methods for concurrent operations on B*-trees have been discussed by Bayer and Schkolnick [3] and others [6, 12, 13]. The solution given in the current paper

This research was supported by the National Science Foundation under Grant MCS76-16604.
Authors' present addresses: P. L. Lehman, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213; S. B. Yao, Department of Computer Science and College of Business and Management, University of Maryland, College Park, MD 20742.

ACM Transactions on Database Systems, Vol. 6, No. 4, December 1981, Pages 650–670.

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:

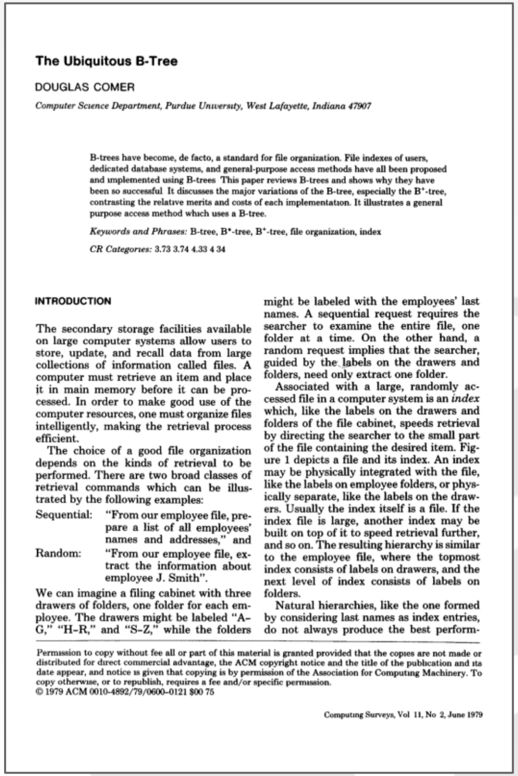
→ **B-Tree** (1971)
→ **B+Tree** (1973)
→ **B*Tree** (1977?)
→ **B^{link}-Tree** (1981)

B+TREE

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

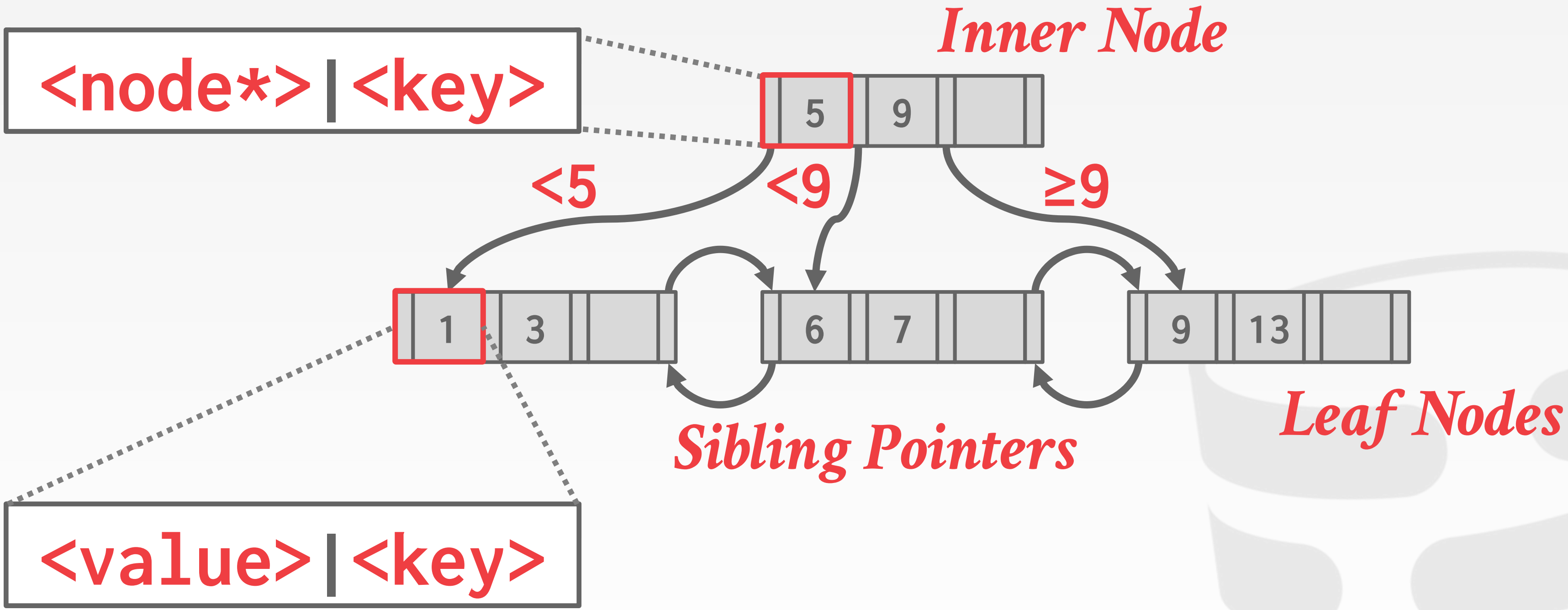
Natural hierarchies, like the one formed by considering last names as index entries, do not always produce the best perform-

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 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

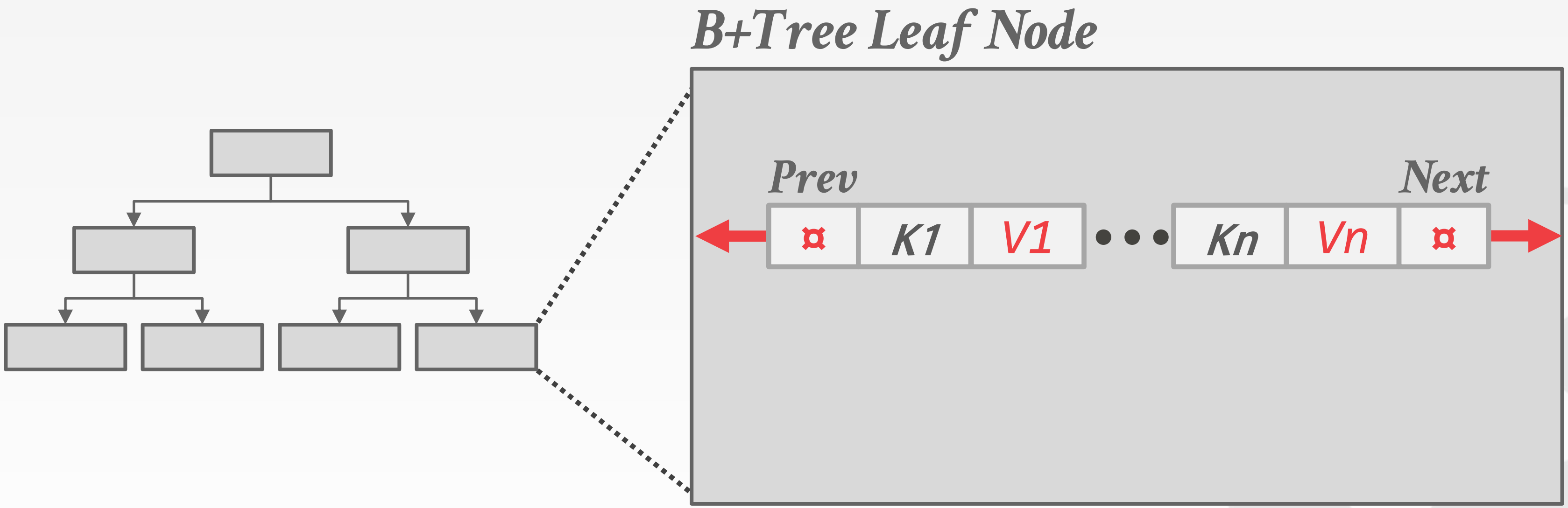
Every B+Tree node is comprised of an array of key/value pairs.

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

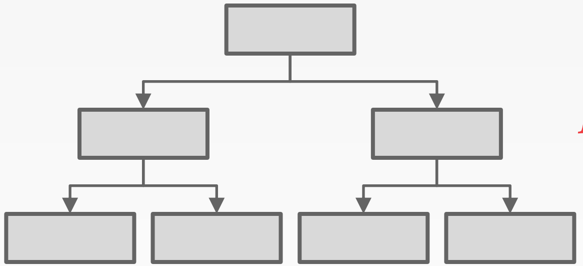
→ The values will differ based on whether the node is classified as **inner nodes** or **leaf nodes.**

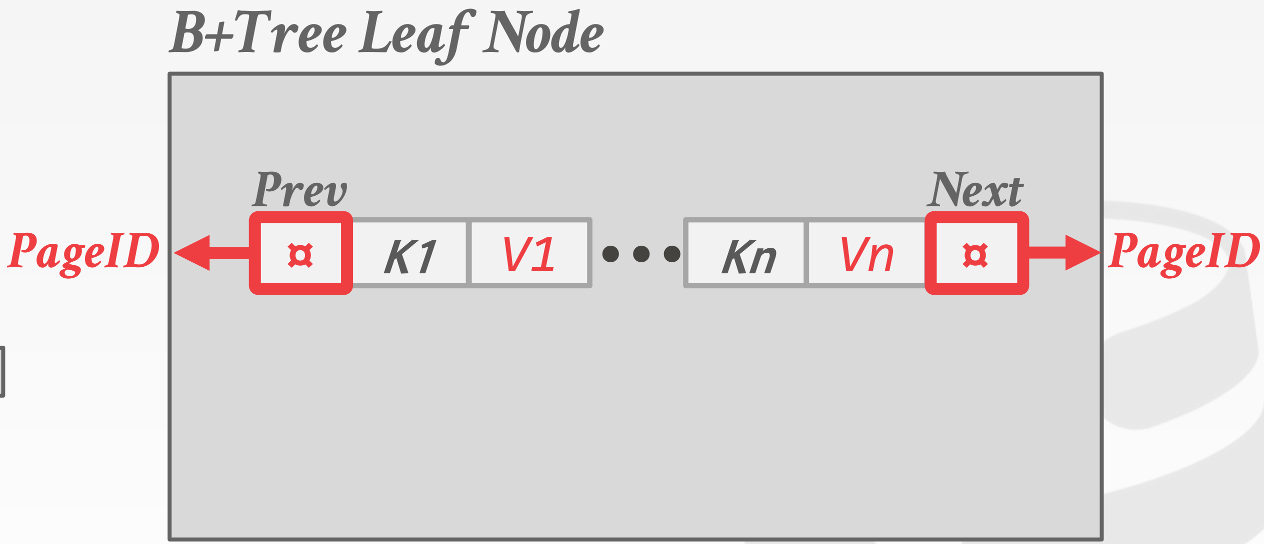
The arrays are (usually) kept in sorted key order.

B+TREE LEAF NODES

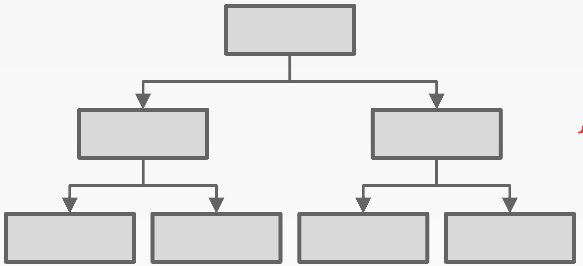


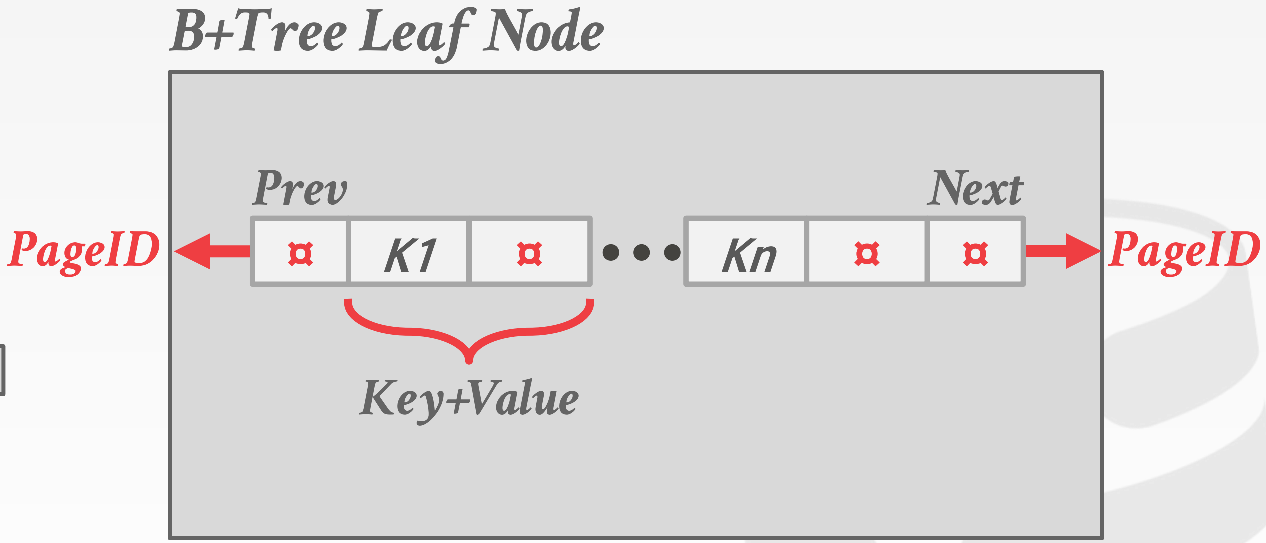
B+TREE LEAF NODES



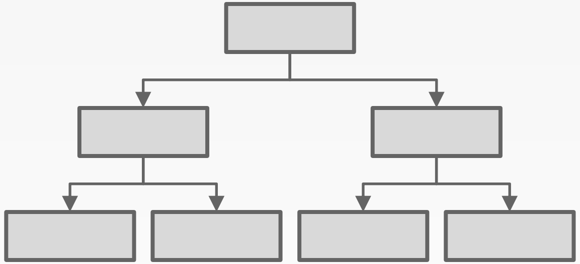


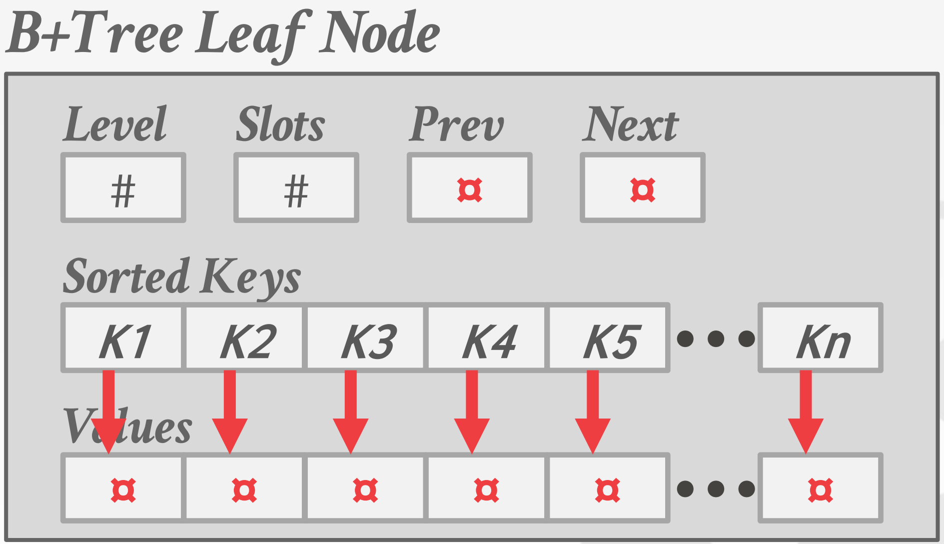
B+TREE LEAF NODES





B+TREE LEAF NODES





LEAF NODE VALUES

Approach #1: Record Ids

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

Approach #2: Tuple Data

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

→ Secondary indexes must store the record id as their values.

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.

SELECTION CONDITIONS

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

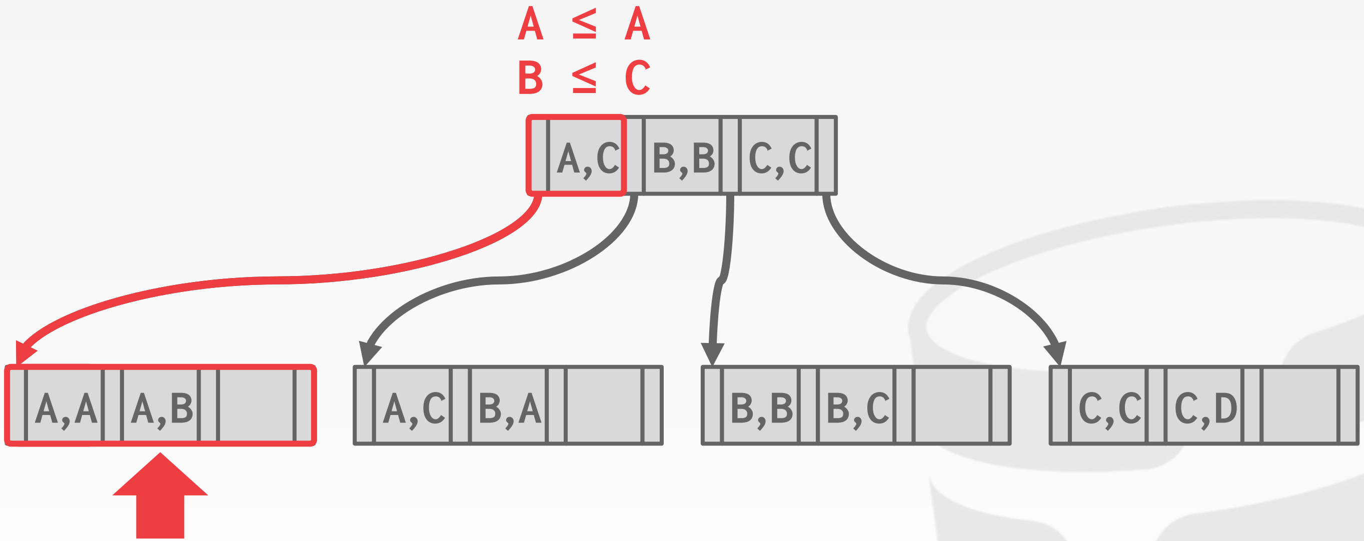
Example: Index on `<a,b,c>`
→ Supported: `(a=5 AND b=3)`
→ Supported: `(b=3)`.

Not all DBMSs support this.

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

SELECTION CONDITIONS

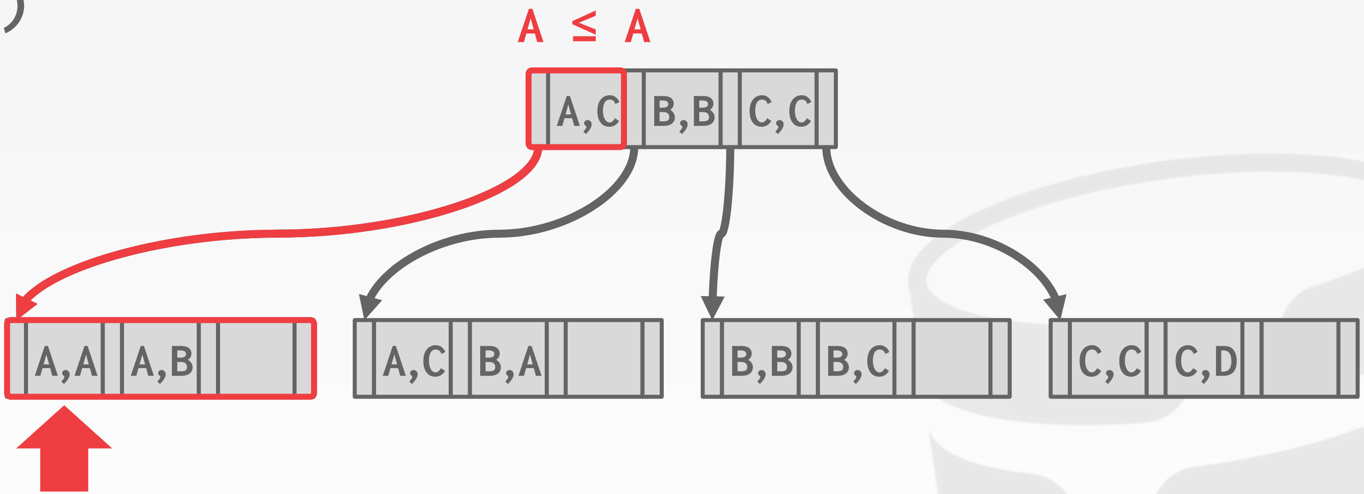
Find Key=(A,B)



SELECTION CONDITIONS

Find Key=(A,B)

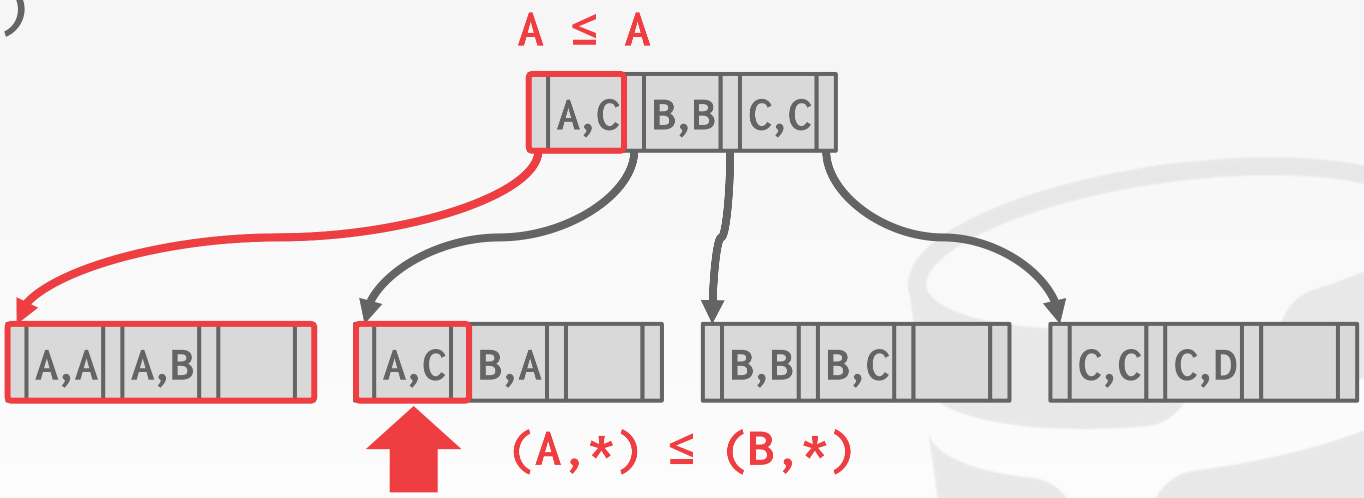
Find Key=(A,*)



SELECTION CONDITIONS

Find Key=(A,B)

Find Key=(A,*)

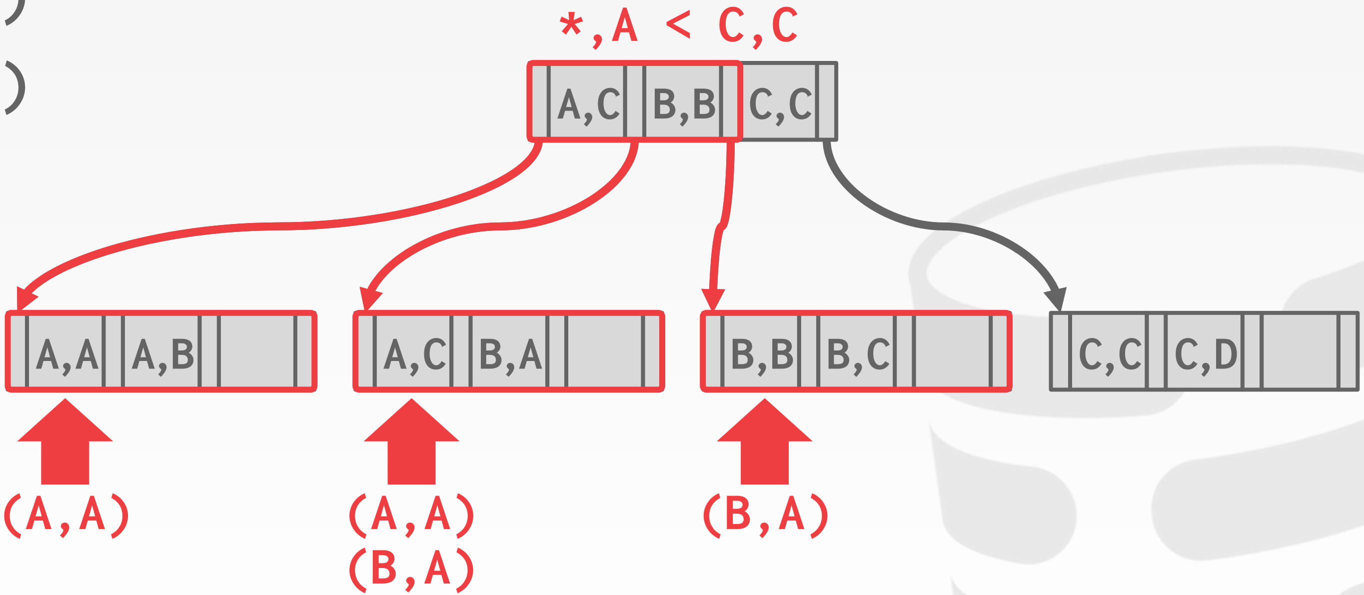


SELECTION CONDITIONS

Find Key=(A,B)

Find Key=(A,*)

Find Key=(*,A)



B+TREE – INSERT

Find correct leaf node **L**.
Put 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 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+TREE – DUPLICATE KEYS

Approach #1: Append Record Id

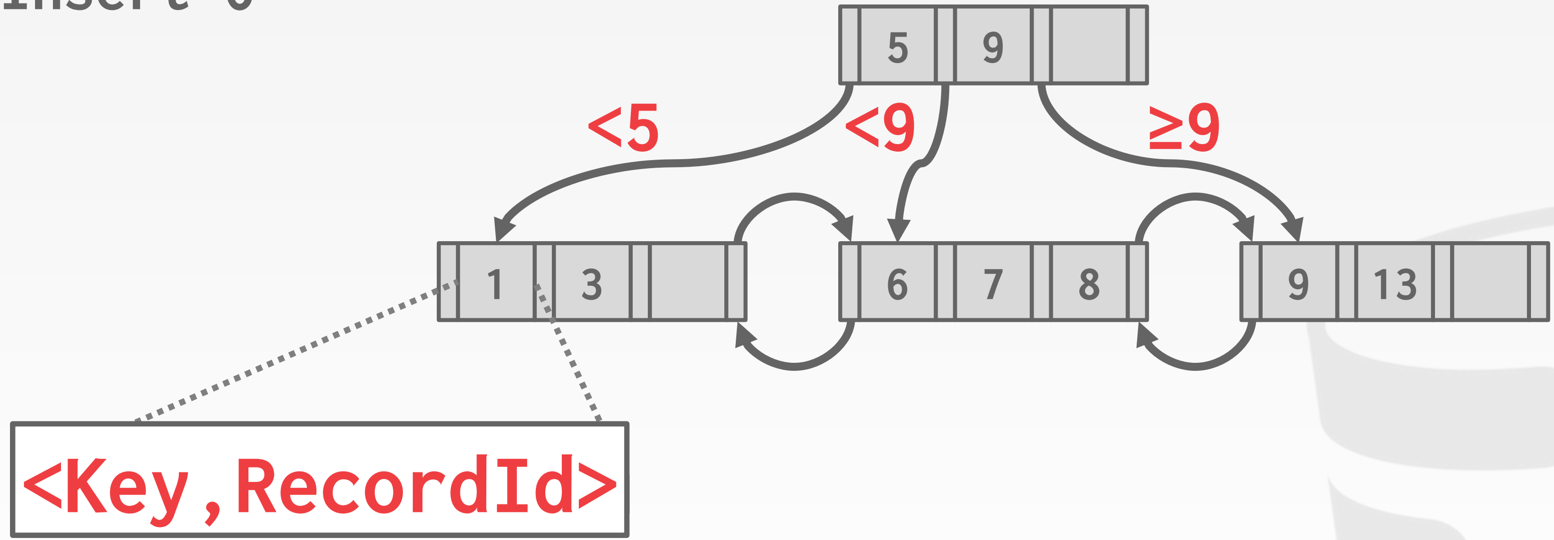
→ Add the tuple's unique record id as part of the key to ensure that all keys are unique.
→ The DBMS can still use partial keys to find tuples.

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.

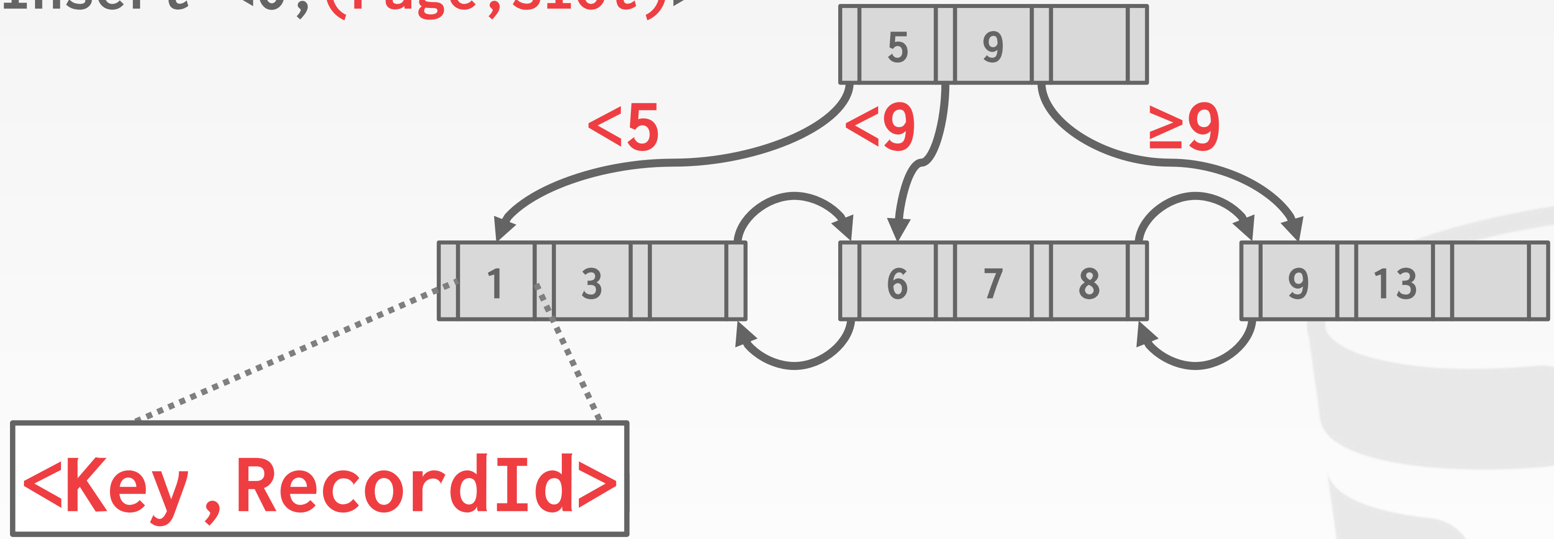
B+TREE – APPEND RECORD ID

Insert 6



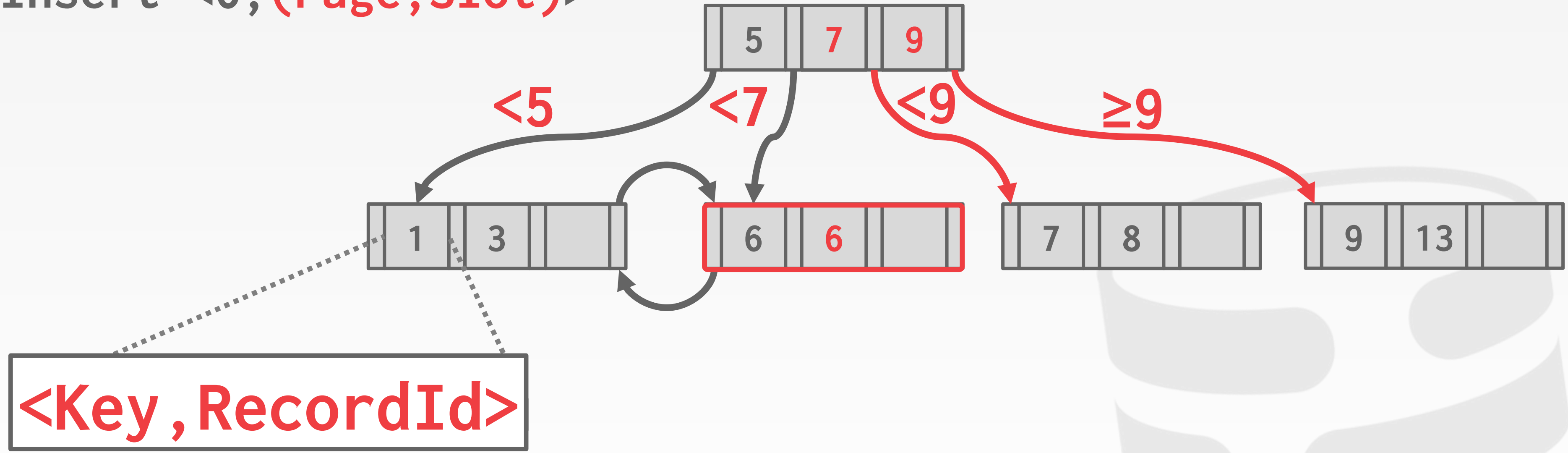
B+TREE – APPEND RECORD ID

Insert <6,(Page,Slot)>



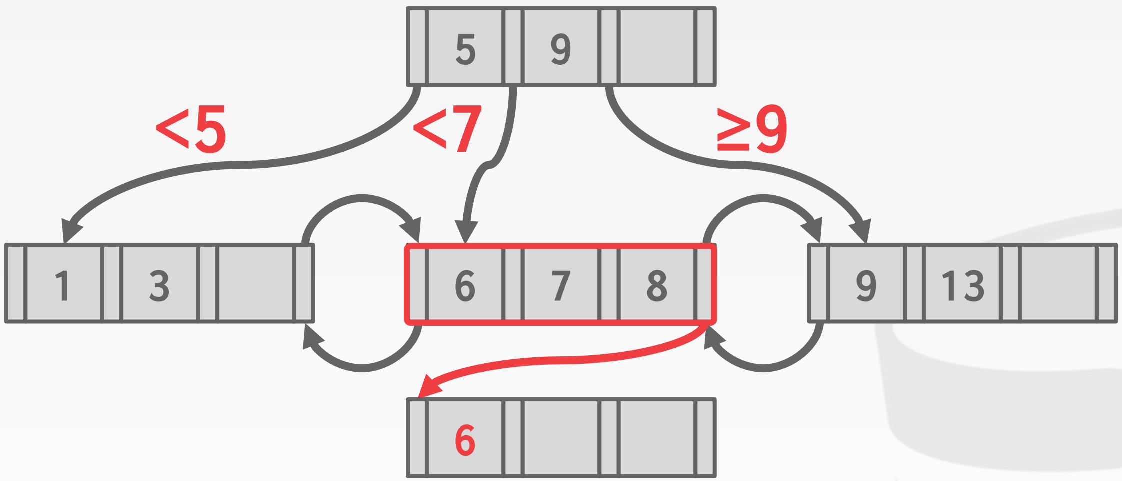
B+TREE – APPEND RECORD ID

Insert <6,(Page,Slot)>



B+TREE – OVERFLOW LEAF NODES

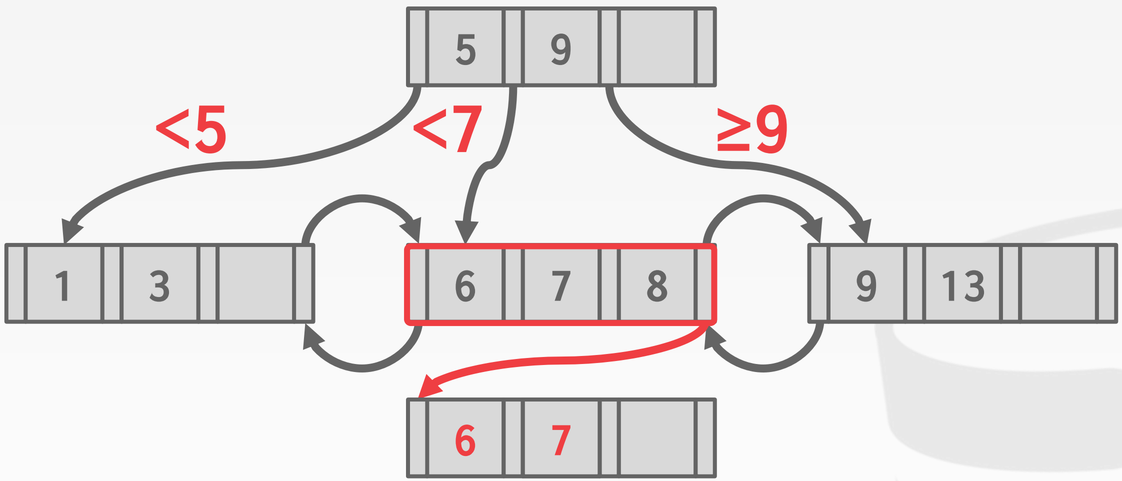
Insert 6



B+TREE – OVERFLOW LEAF NODES

Insert 6

Insert 7

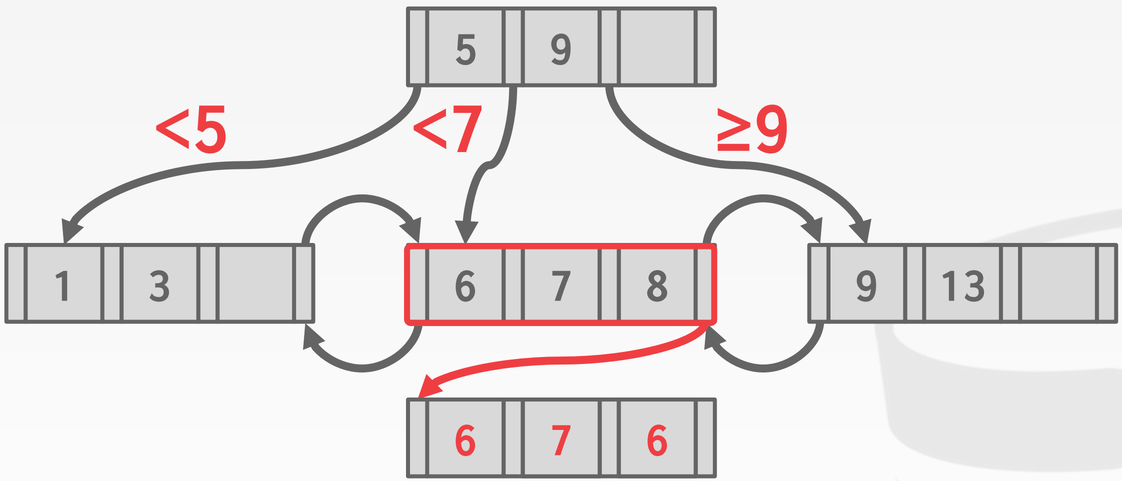


B+TREE – OVERFLOW LEAF NODES

Insert 6

Insert 7

Insert 6



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.

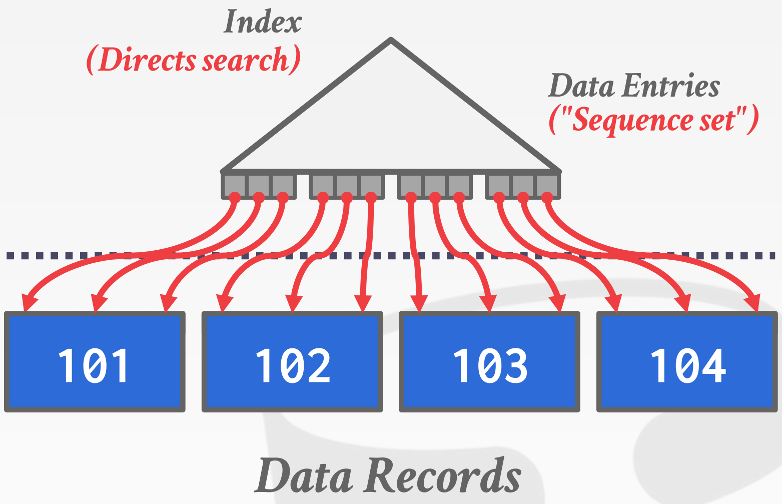
- → If a table does not contain a primary key, the DBMS will automatically make a hidden row id primary key.

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.

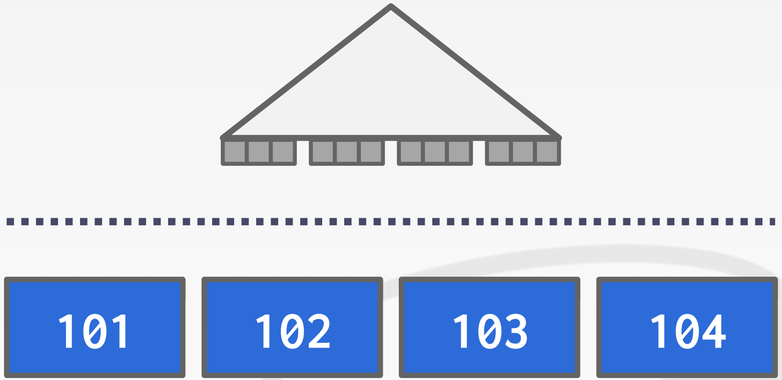
This will always better than external sorting.



HEAP CLUSTERING

Tuples are sorted in the heap's pages using the order specified by a clustering index.

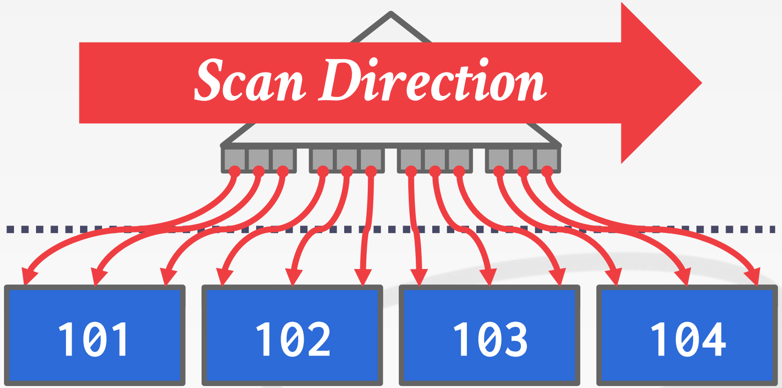
If the query accesses tuples using the clustering index's attributes, then the DBMS can jump directly to the pages that it needs.



HEAP CLUSTERING

Tuples are sorted in the heap's pages using the order specified by a <u>clustering index</u>.

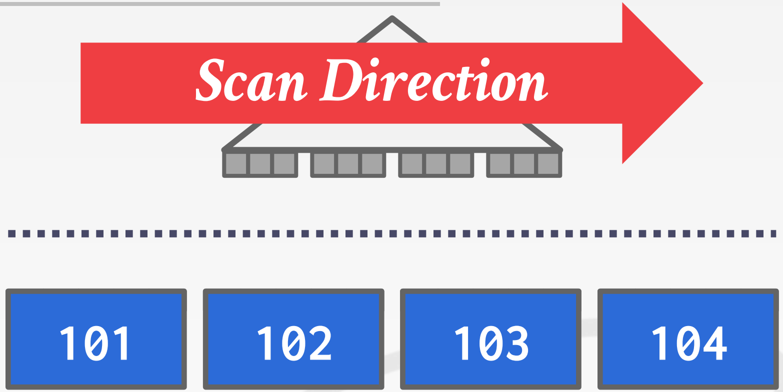
If the query accesses tuples using the clustering index's attributes, then the DBMS can jump directly to the pages that it needs.



INDEX SCAN PAGE SORTING

Retrieving tuples in the order that appear in an unclustered index is inefficient.

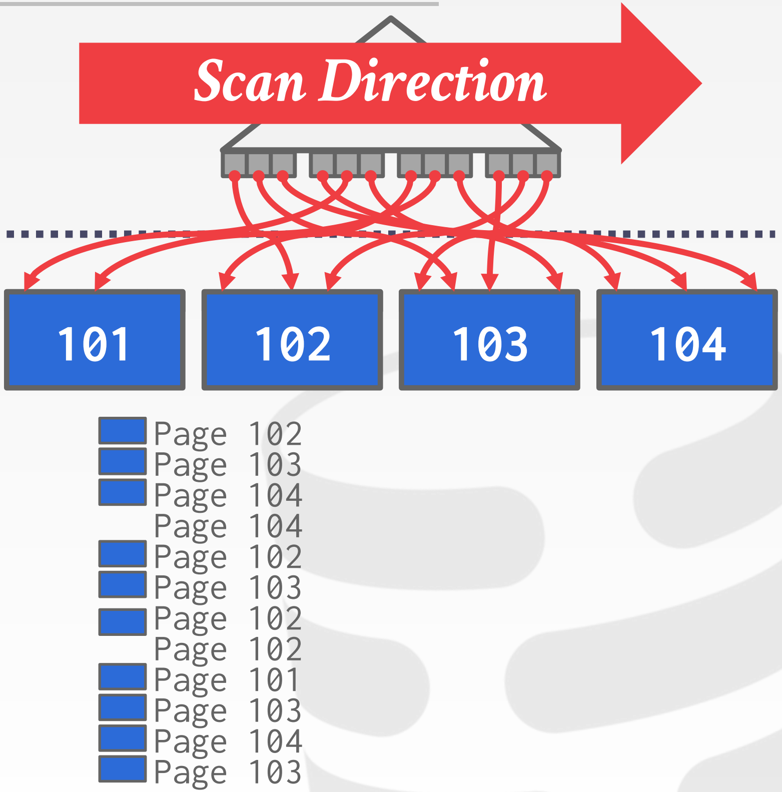
The DBMS can first figure out all the tuples that it needs and then sort them based on their page id.



INDEX SCAN PAGE SORTING

Retrieving tuples in the order that appear in an unclustered index is inefficient.

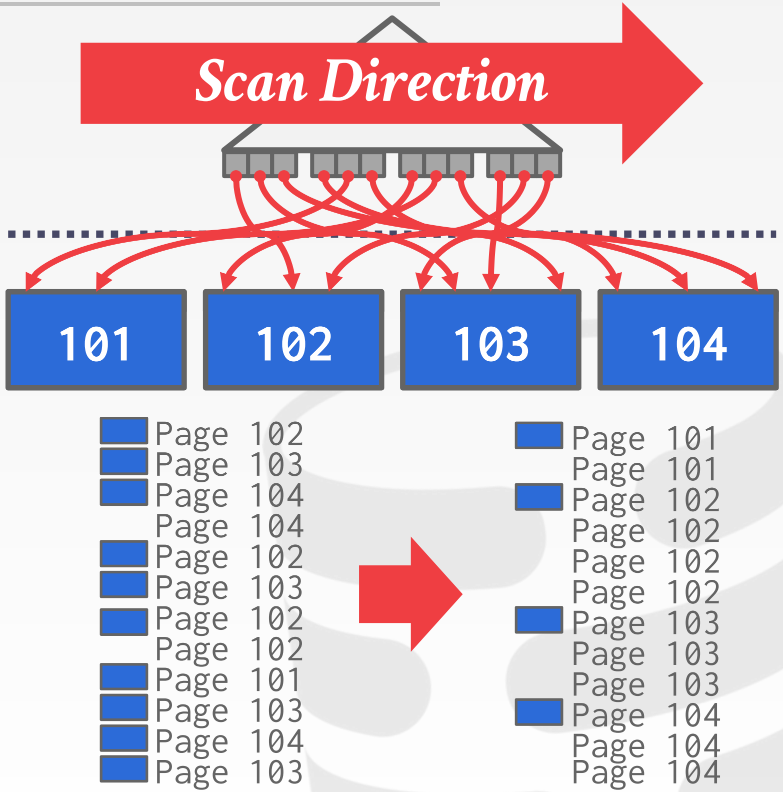
The DBMS can first figure out all the tuples that it needs and then sort them based on their page id.



INDEX SCAN PAGE SORTING

Retrieving tuples in the order that appear in an unclustered index is 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

CONCLUSION

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

NEXT CLASS

More B+Trees

Tries / Radix Trees

Inverted Indexes