Trees (Part 1)

Recap

Hash Tables

- Hash tables are fast data structures that support O(1) look-ups
- Used all throughout the DBMS internals.
  - Examples: Page Table (Buffer Manager), Lock Table (Lock Manager)
- Trade-off between speed and flexibility.

Limitations of Hash Tables

- Hash tables are usually **not** what you want to use for a indexing tables
  - Lack of ordering in widely-used hashing schemes
  - Lack of locality of reference $\longrightarrow$ more disk seeks
  - Persistent data structures are much more complex (logging and recovery)
  - Reference

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 based a subset of those attributes.
- Example: {**Employee Id**, **Dept Id**} $\longrightarrow$ Employee Tuple Pointer
- The DBMS ensures that the contents of **the table** and **the indices** are 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
- B+Tree in Practice
- Design Decisions
- Optimizations

B+Tree Overview

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?)
  - **Blink-Tree** (1981)

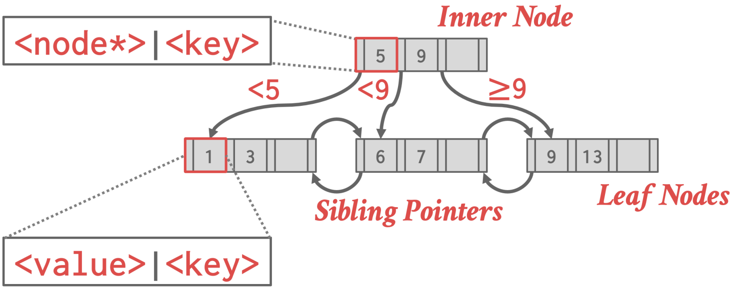
B+Tree

- 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 disk storage (*i.e.*, read and write at page-granularity).

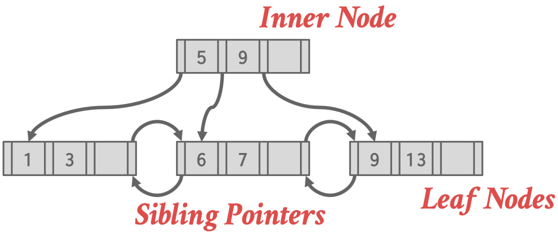
B+Tree Properties

- 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 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 (**node pointers**)

B+Tree Example



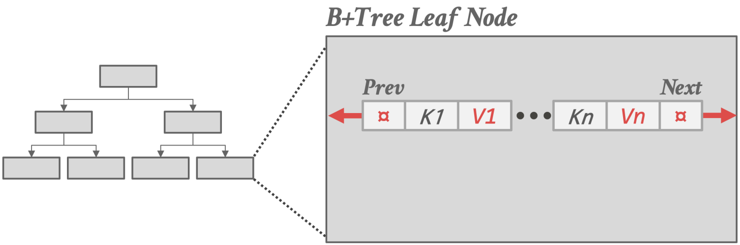
B+Tree Example



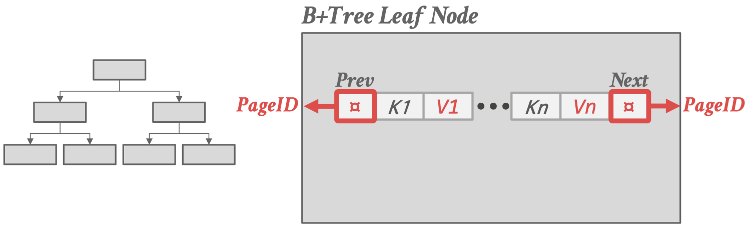
Nodes

- 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.
  - Inner nodes: Values are pointers to other nodes.
  - Leaf nodes: Values are pointers to tuples or actual tuple data.
- The arrays are (usually) kept in sorted key order.

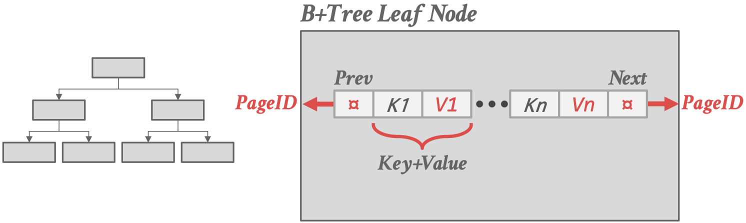
B+Tree Leaf Nodes



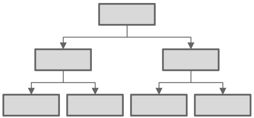
B+Tree Leaf Nodes

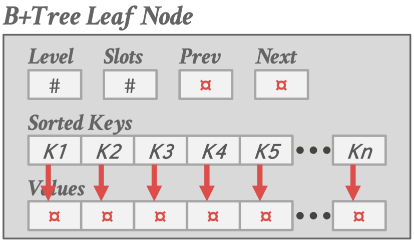


B+Tree Leaf Nodes



B+Tree Leaf Nodes





Node

```
struct Node {
    /// The level in the tree.
    uint16_t level;
    /// The number of children.
    uint16_t count;
        ...
};

void print_node(Node *node);
```

Node

```
struct InnerNode: public Node {
        /// The capacity of a node.
    static constexpr uint32_t kCapacity = 42;
    /// The keys.
    KeyT keys[kCapacity];
    /// The children.
    uint64_t children[kCapacity];
        ...
};
```

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** typically 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.
- Easier to support concurrent index access when only values are stored in leaf nodes.

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.
- Splits help grow the tree by one level

B+Tree: Visualization

- Demo
- 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.

B+Tree In Practice

B+Tree Statistics

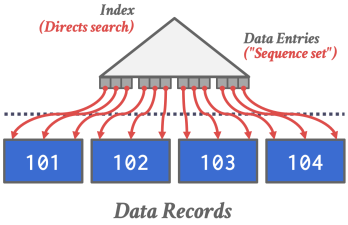
- Typical Fill-Factor: 67
- Pages per level:
  - Level 1 = 1 page = 8 KB
  - Level 2 = 134 pages = 1 MB
  - Level 3 = 17,956 pages = 140 MB

Data Organization

- A table can be stored in two ways:
  - **Heap-organized storage**: Organizing rows in **no particular order**.
  - **Index-organized storage**: Organizing rows in **primary key order**.
- Types of indexes:
  - **Clustered index**: Organizing rows in a **primary key order**.
  - **Unclustered index**: Organizing rows in a **secondary key order**.

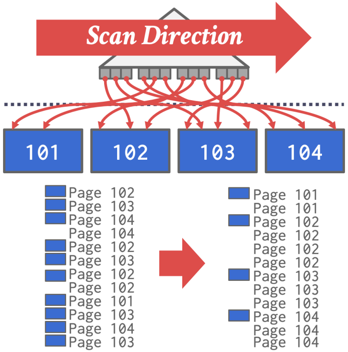
Clustered Index

- Tuples are kept sorted on disk using the order specified by **primary key**.
- If the query accesses tuples using the clustering index's attributes, then the DBMS can jump directly to the pages that it needs.
- Traverse to the left-most leaf page, and then retrieve tuples from all leaf pages.



Unclustered Index

- 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.



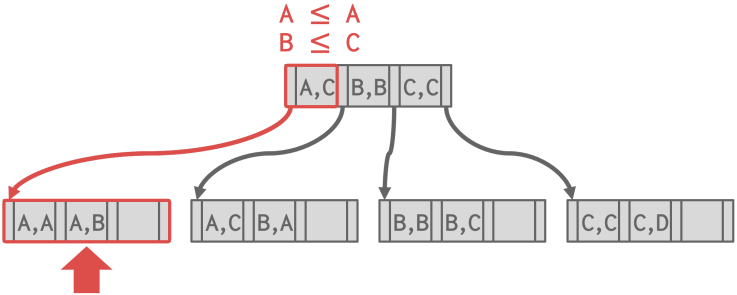
Clustered vs. Unclustered Index

- Clustered index
  - Only one clustered index per table
  - Example: {**Employee Id**} $\longrightarrow$ **Employee Tuple Pointer**
- Unclustered index
  - Multiple unclustered indices per table
  - Example: {**Employee City**} $\longrightarrow$ **Clustered Index Pointer** or **Employee Tuple Pointer**
  - Accessing data through a non-clustered index may need to go through an extra layer of indirection

Filtering Tuples

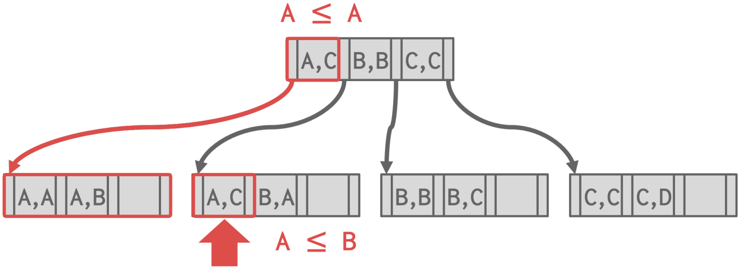
- The DBMS can use a B+Tree index if the filter uses any of the attributes of the key.
- Example: Index on **<a,b,c>**
  - Supported: (a=5 AND b=3)
  - Supported: (b=3).
- For hash index, we must have **all attributes** in search key.

Filtering Tuples



Find Key=(A,B)

Filtering Tuples



Find Key=(A,*)

B+Tree Design Decisions

B+Tree Design Decisions

- Node Size
- Merge Threshold
- Variable Length Keys
- Non-Unique Indexes
- Intra-Node Search
- Modern B-Tree Techniques

Node Size

- The slower the storage device, the larger the optimal node size for a B+Tree.
  - HDD ~1 MB
  - SSD: ~10 KB
  - In-Memory: ~512 B
- Optimal sizes varies depending on the workload
  - Leaf Node Scans (OLAP) vs. Root-to-Leaf Traversals (OLTP)

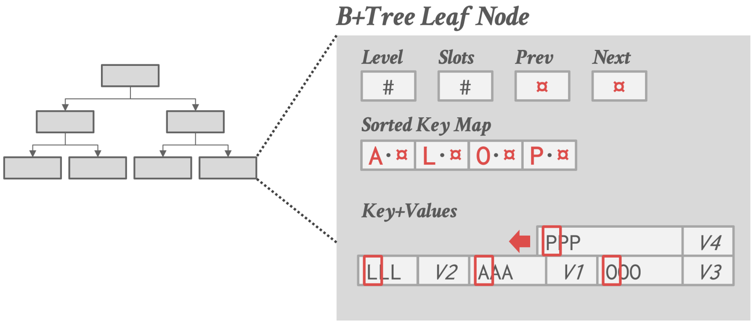
Merge Threshold

- Some DBMSs do not always merge nodes when it is half full.
- Delaying a merge operation may reduce the amount of reorganization.
- It may also 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 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.

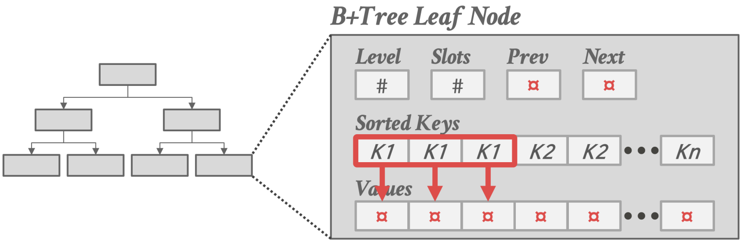
Variable Length Keys: Key Map



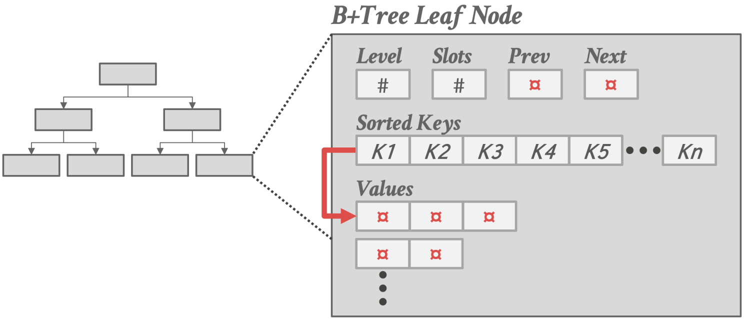
Non-Unique Indexes

- **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 Indexes: Duplicate Keys

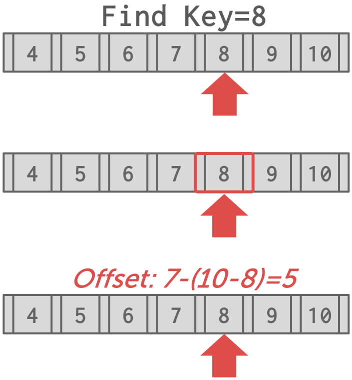


Non-Unique Indexes: Value Lists



Intra-Node Search

- **Approach 1: Linear Search**
  - Scan node keys from beginning to end.
- **Approach 2: Binary Search**
  - Jump to middle key, pivot left/right depending on comparison.
- **Approach 3: Interpolation Search**
  - Approximate location of desired key based on known distribution of keys.



Intra-Node Search

```
struct InnerNode: public Node {
    std::pair<uint32_t, bool> lower_bound(const KeyT &key) {
      /// Set lower and upper bounds for binary search
      uint16_t l = 0;
      uint16_t h = this->count - 2;
    }
        ...
};
```

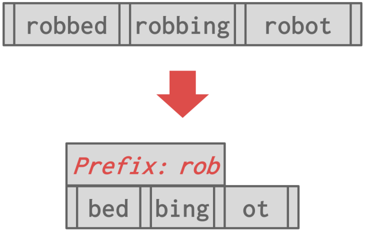
Optimizations

Optimizations

- Prefix Compression
- Suffix Truncation
- Bulk Insert
- Pointer Swizzling

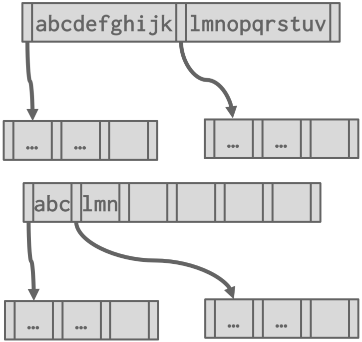
Prefix Compression

- 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.



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.



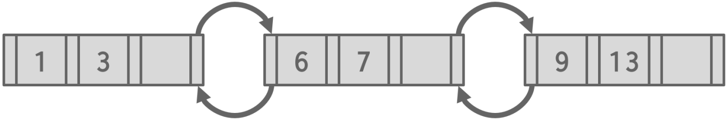
Bulk Insert

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

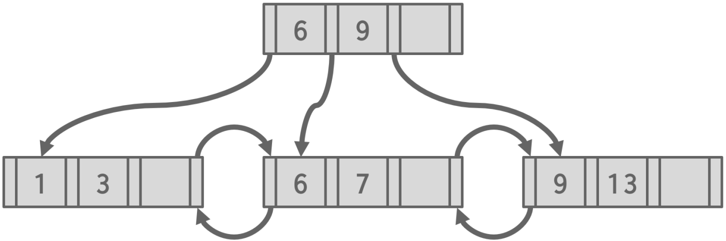
Keys: 3, 7, 9, 13, 6, 1

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

Bulk Insert



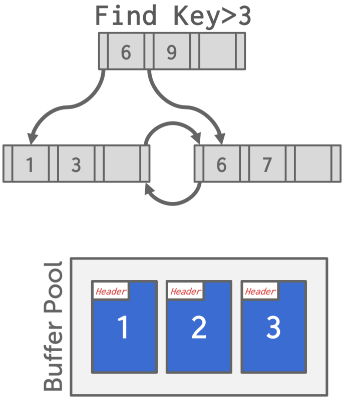
Bulk Insert

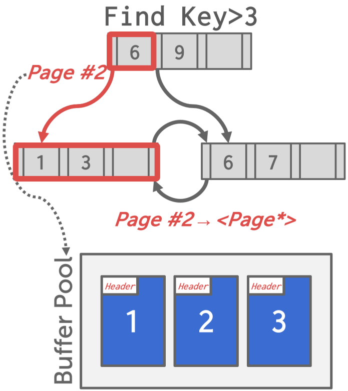


Pointer Swizzling

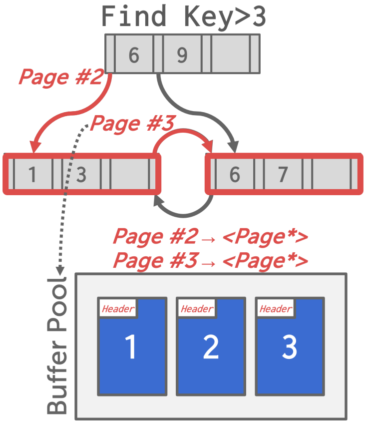
- Nodes use **page ids** to reference other nodes in the index.
- The DBMS must get the memory location from the page table during traversal.
- If a page is pinned in the buffer pool, then we can store **raw pointers** instead of page ids.
- This **avoids address lookups** from the **page table**.

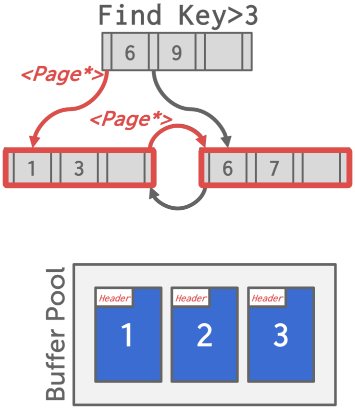
Pointer Swizzling





Pointer Swizzling





Conclusion

Conclusion

- The venerable B+Tree is always a good choice for your DBMS.
- Next Class
  - More B+Trees
  - Tries / Radix Trees
  - Inverted Indexes