Relational Database System Implementation

CS122 – Lecture 11

Winter Term, 2018-2019

Indexes

- Many queries only need a small number of records
  - Records with a specific value
  - Records with a specific range of values
- Most queries involve join operations
  - Correlate values of a column across two or more tables
- So far we have used simple file scans
  - Prohibitively slow for large data sets

- Better databases use *indexes* to speed access to records with specific values

Indexes (2)

- An index is a separate access structure associated with a particular table
  - e.g. tables and their indexes are usually stored in separate files
  - Much smaller, and structured for faster lookups
- Each index has an associated *search key*
  - Attribute (or set of attributes) used to look up records
  - This kind of "key" is completely separate from primary keys, candidate keys, etc.
- A table can have multiple indexes
  - Each index will have its own search key

Indexes (3)

- Several kinds of indexes with different capabilities:
- Access patterns and access time
  - Types of access that are supported efficiently
  - Time it takes to access a particular item or set of items
- Indexes must be kept in sync with their table
  - Time it takes to insert a new data item
  - Time it takes to delete a data item
- Indexes also consume extra space!
  - Additional space overhead taken by the index
  - Usually, extra space taken by index is far outweighed by the performance improvement

Index Types

- Two main categories of indexes
- *Ordered indexes* maintain a sorted ordering based on search key values
  - Logarithmic time for finding a specific record, or a boundary of a range
  - Can retrieve values in search key order
- *Hash indexes* use a hash function to distribute search key values across buckets
  - Constant time for finding a specific record, or a group of records with same value
  - Very inefficient for retrieving a range of values

Sequential Files and Indexes

- Sequential files are also stored in search key order
- An index on the search key can still be useful!
  - An index lookup can be much faster than doing a binary search on the table itself
    - Index entries are much smaller than tuples
    - e.g. 2-3 block reads, vs. 10+ block reads
- *Primary indexes*:
  - Ordered indexes that are in the same search-key order as their associated tables
  - Also called *clustering indexes*
  - (Has nothing to do with primary keys!!!)
- Sequential file + primary index = *index-sequential file*

Dense and Sparse Indexes

- For a sequential file with a primary index, the index can be either dense or sparse
- *Dense indexes* store an entry for every distinct value in the search key
  - Easy to find any particular value; all are represented in index
  - Index can easily become very large, for large tables
- *Sparse indexes* only store entries for a subset of the values in the search key
  - To find a specific record, find index entry with largest value less than desired value
  - Then, scan through sequential file from that location, until the record is found

Secondary Indexes

- *Secondary indexes* don't share the same search key as their associated table
  - Table may have a different search key order
  - Table may be a heap file with no specific order!
- Secondary indexes *must* be dense
  - Must include an entry for every value of search key
  - Must include a pointer to every record in the table
  - Since table is in a different order from the index, the index won't be generally useful if it isn't dense

B-Tree Indexes

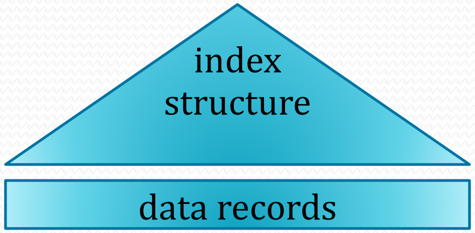
- Most widely used index structure is the *B-tree* family of index structures
  - A *multilevel* indexing structure built as a balanced tree
  - Supports <u>both</u> sequential access and direct access!
- Depth of tree grows automatically as required by the table being indexed
- Space within disk blocks is managed automatically; all blocks at least 50% full, no overflow needed *(usually)*
- Branching factor is *very* large (normally hundreds), producing an extremely broad, flat tree
  - Disk accesses required is proportional to *depth* of tree

B-Tree Indexes (2)

- Not clear what the "B" stands for in B-trees…
  - Definitely not "binary" – these are multiway trees
  - "Balanced," "broad," "bushy" have all been suggested
  - Developed by "Bayer" (and McCreight) while at "Boeing"
  - *Who knows… (Who cares?)*
- Different versions vary in rather important ways:
  - How full are tree-nodes allowed to get before splitting?
  - Is indexing and storage kept together or separate?
- Of all B-tree variants, most widely used is B^+-tree
  - When people say "B-tree", they usually mean B^+-tree

B⁺-Tree Indexes

- B⁺-trees separate indexing structure and data records
  - Original B-tree structure mixes these!



- Main implication:
  - Internal nodes have different structure than leaf nodes
  - Internal nodes only store keys *(plus structural data)*
  - Leaf nodes store keys and data records as well
- B⁺-trees (and other variants) can be used for storing sequential files as well as for indexes
  - In indexes, "records" are simply file-pointers into table

B^+-Tree Indexes (2)

- Other relevant details:
  - All tree-nodes must be at least 50% full (except for root)
  - Every path from root to leaf is the same length
  - Key-values may be repeated in different tree-nodes (original B-tree eliminates this redundancy, but mixes the indexing and data records)
- B^+-trees are often used for filesystems
  - Index built on top of sequential file laid out on disk
  - Allows rapid mapping of logical file-location to physical cylinder/sector on disk
  - Also facilitates sequential access of file contents

B⁺-Tree Nodes

- Tree nodes have up to *n* children
  - A simplification, for now: *n* is fixed for an entire tree
    - Value of *n* depends on block size, key size, and pointer size
    - Can often be large, e.g. a few hundred!
- A node stores *n* pointers and *n* – 1 values

P_1	K_1	P_2	K_2	P_3	…	P_{n-1}	K_{n-1}	P_n

  - K_i are search-key values
  - P_i are pointers that specify the tree's structure
  - Key values are kept in sorted order: if $i < j$ then $K_i \le K_j$
    - (In case of duplicate key values, may have neighboring $K_i = K_j$)

B⁺-Tree Leaf Nodes

- For leaf nodes:

P_1	K_1	P_2	K_2	P_3	…	P_{n-1}	K_{n-1}	P_n

  - Pointer P_i refers to a record with search-key value K_i
- If search key is a candidate key, only one record in the table will have the key-value K_i
  - A common case – indexes built on primary keys for enforcing key and referential integrity constraints
  - P_i points to the record with key value K_i

B+-Tree Leaf Nodes (2)

- For leaf nodes:

P_1	K_1	P_2	K_2	P_3	…	P_{n-1}	K_{n-1}	P_n

  - Pointer P_i refers to a record with search-key value K_i
- If search key is not a candidate key, multiple records in the table will have the same key-value K_i
  - Unfortunately, also a common case…
- Two options:
  - Can simply repeat search-key value multiple times
  - Or, have P_i point to a bucket containing pointers for all records with key-value K_i (complicated; adds I/O costs)

B^+-Tree Leaf Nodes (3)

- For leaf nodes:

P_1	K_1	P_2	K_2	P_3	…	P_{n-1}	K_{n-1}	P_n

  - Pointer P_n points to the next leaf-node in the sequence
- Within a node, key values are kept in sorted order
  - (if $i < j$ then $K_i \leq K_j$)
- Leaves contain non-overlapping ranges of key/record associations
- B^+-tree orders leaves in increasing sequential order
  - Allows *very* easy traversal of dataset in search-key order

B^+-Tree Non-Leaf Nodes

- For non-leaf nodes:

P_1	K_1	P_2	K_2	P_3	…	P_{n-1}	K_{n-1}	P_n

  - All pointers P_i refer to other B^+-tree nodes
- For $1 < i < n$:
  - Pointer P_i points to subtree containing search-key values of at least K_{i-1}, but less than K_i
- For $i = 1$ or $i = n$:
  - Pointer P_1 points to subtree with search-key values less than K_1
  - Pointer P_n points to subtree containing search-key values of at least K_{n-1}

B^+-Tree Non-Leaf Nodes (2)

- For non-leaf nodes:

P_1	K_1	P_2	K_2	P_3	…	P_{n-1}	K_{n-1}	P_n

  - All pointers P_i refer to other B^+-tree nodes
- In other words:
  - P_1 points to subtree with search-keys in range $[-\infty, K_1)$
  - P_2 points to subtree with search-keys in range $[K_1, K_2)$
  - P_3 points to subtree with search-keys in range $[K_2, K_3)$
  - …
  - P_{n-1} points to subtree with search-keys in range $[K_{n-2}, K_{n-1})$
  - P_n points to subtree with search-keys in range $[K_{n-1}, +\infty)$

Non-Full B^+-Tree Nodes

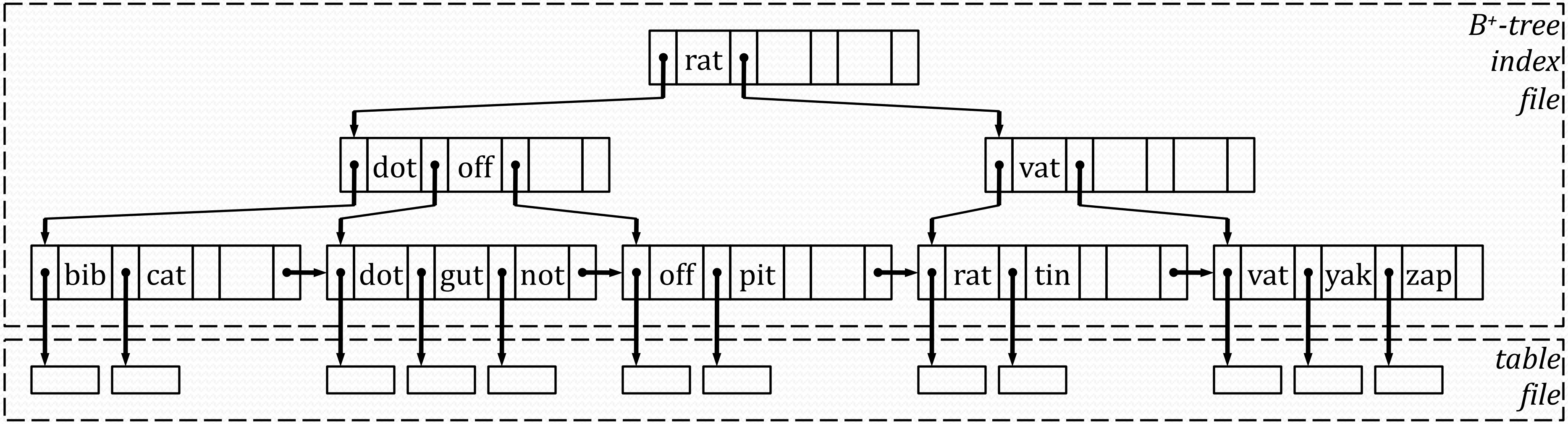
- B^+-tree nodes must be at least 50% full
  - Specified in terms of n, number of pointers in each node
  - (Can also state this constraint as number of bytes used)
- The root node is not required to be at least 50% full
  - *(Often simply don't have enough data to enforce this.)*
- Non-leaf nodes must have at least $\lceil n/2 \rceil$ pointers
  - Must contain at least $\lceil n/2 \rceil - 1$ keys
  - e.g. for tree with $n = 5$:
    - $\lceil n/2 \rceil = 3$ ptrs and 2 keys, minimum

Non-Full B$^+$-Tree Nodes (2)

- Leaf nodes always use P_n to point to next leaf-node…
- Don't count this "next leaf-node" pointer in the measure of whether a leaf is half-full
  - Each P_i points to a row with value K_i
  - Must have at least $\lceil (n - 1)/2 \rceil$ pointers and keys
  - e.g. tree with $n = 4$:
    - $\lceil (n - 1)/2 \rceil = 2$ ptrs and 2 keys, minimum

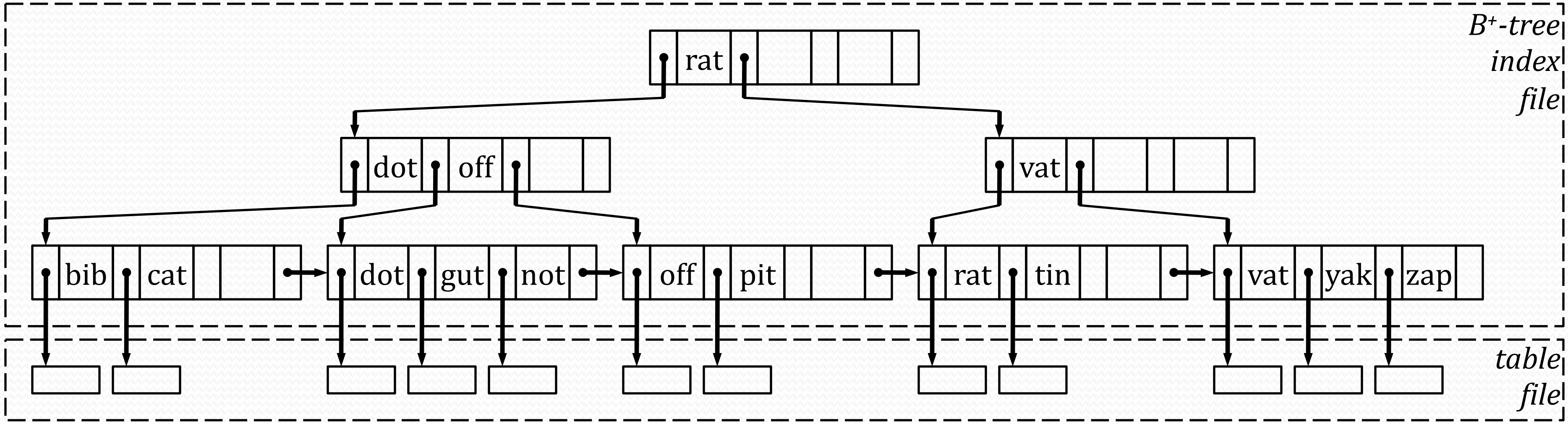
Example B+-Tree

- Will use a tree with low *n* for sake of simplicity
  - Easy to comprehend
  - Will provoke frequent need to split and join nodes
- A simple tree with *n* = 4:
  - Non-leaf nodes must have at least 2 pointers and 1 key
  - Leaf nodes must have at least 2 pointers and 2 keys



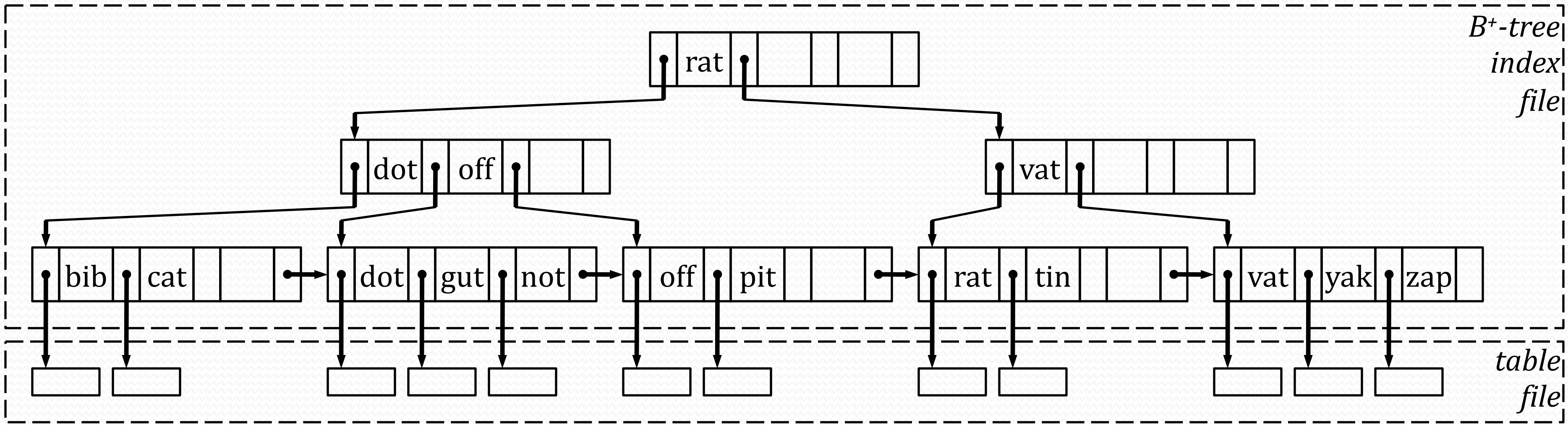
Example B+-Tree (2)

- Also specify that search-key values are unique
  - Don’t need to worry about runs of entries with the same search-key value. (We’ll handle this later.)
- Finally, specify that this is a dense index
  - Every single value in table also appears in the index
  - No additional search needed once we reach leaf record



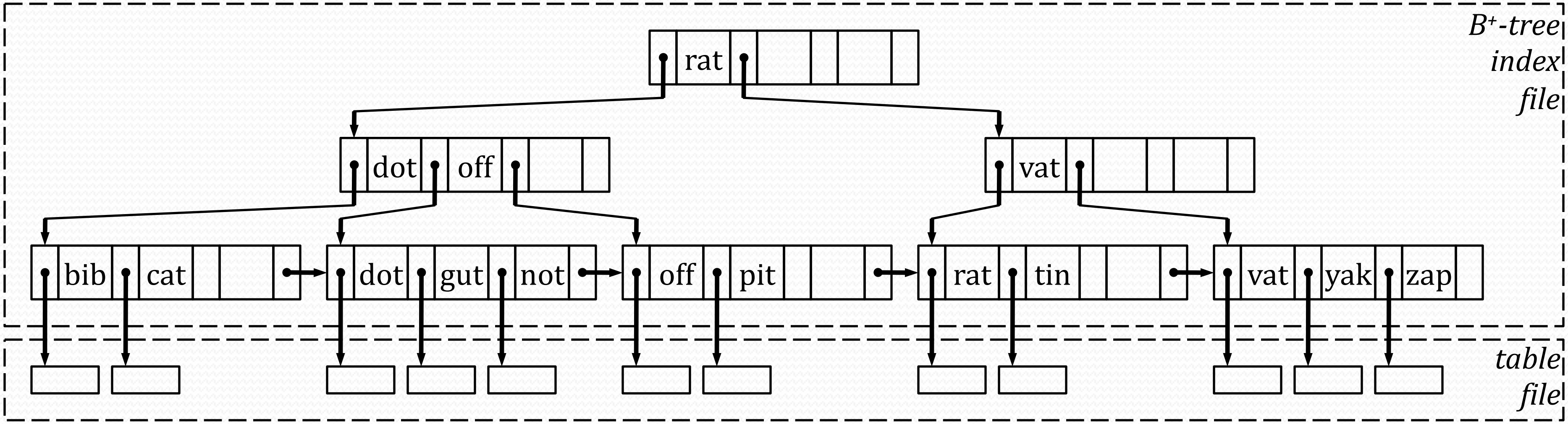
B$^+$-Trees: Querying

- Look up the record with the search-key value *V*
- Given the value *V*, can follow tree structure to find the exact leaf-node where *V* should be stored
  - If *V* isn't in that leaf node, then *V* isn't in the index



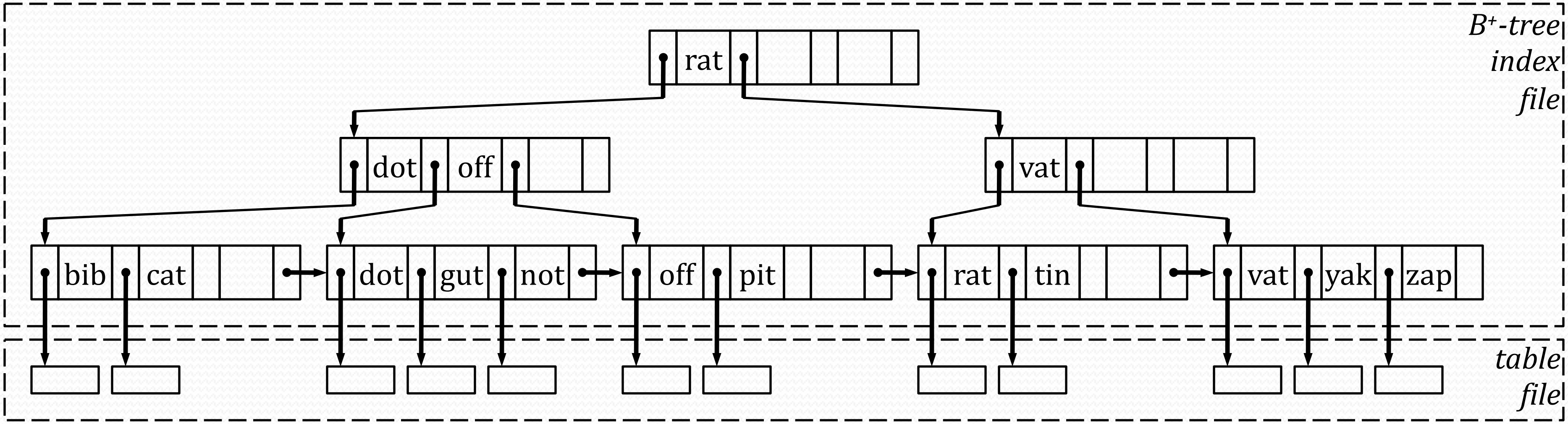
B⁺-Trees: Querying (2)

- Navigate non-leaf nodes separately from leaf-node
- Each non-leaf node has m pointers, P_1 .. P_m ($1 < m \leq n$)
- For a given non-leaf node, start with $i = 1$:
  - If $V < K_i$, follow pointer P_i
  - If $V = K_i$, follow pointer P_{i+1}
  - If $i + 1 < m$, increment i and repeat; otherwise follow P_m



B⁺-Trees: Querying (3)

- Once we reach a leaf node, it's easy
- Find K_i that equals V; P_i points to record with value V
- If node doesn't contain any K_i that equals V, then the table simply doesn't contain a record with value V
  - Don't need to go to next leaf-node, or anything like that



B^+-Trees: Querying (4)

- Algorithm to find record with search-key value V:

  C = root node
  while C is a non-leaf node:
      m = number of pointers in C; $i = 1$
    SearchNode:
      if $V < K_i$ then set $C = C.P_i$
      else if $V = K_i$ then set $C = C.P_{i+1}$
      else if $i + 1 < m$ then i++; goto SearchNode
      else set $C = C.P_m$

  /* Now, C is a leaf node */
  Iterate over all K_i in leaf-node C:
    if $V = K_i$ then return P_i
  If no K_i found then return *null*

"Go Right On Equality!"

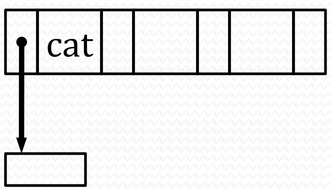
- For non-leaf nodes:

P_1	K_1	P_2	K_2	P_3	…	P_{n-1}	K_{n-1}	P_n

  - All pointers P_i refer to other B^+-tree nodes
- Structural rules:
  - P_1 points to subtree with search-keys in range $[-\infty, K_1)$
  - P_2 points to subtree with search-keys in range $[K_1, K_2)$
  - …
- Specifically, if we are looking for search-key value V:
  - If $K_i = V$, follow pointer to the <u>right</u> of K_i
  - Some B^+-tree impls. handle this case by going left
  - *(Always pay attention to the implementation details…)*

B^+-Trees: Insertion

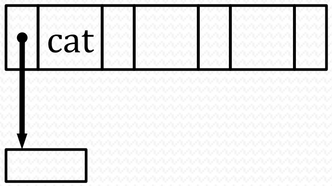
- Insertion is easy, except when a node overflows
  - Since *n* is generally large, overflows occur infrequently
- Simplest case: inserting into an empty B^+-tree index
  - In this case, the root node will also be a leaf node
- Example: Insert "cat" into empty index



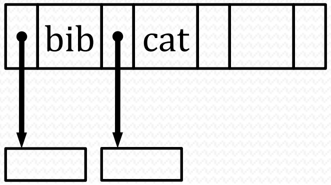
- Note that the leaf-node is < 50% full
  - Simply don't have enough data to satisfy requirement
  - Since it's also the root node, we don't mind

B⁺-Trees: Insertion (2)

- Similarly, inserting other records into a single-node B⁺-tree is easy, as long as there is room in the node
- Example: Insert “bib” into our index
  - B⁺-tree before insertion:

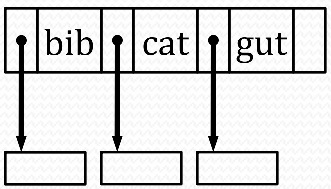


- Must keep K_i values in increasing order…
  - Slide “cat” over in the node, to make room for “bib”

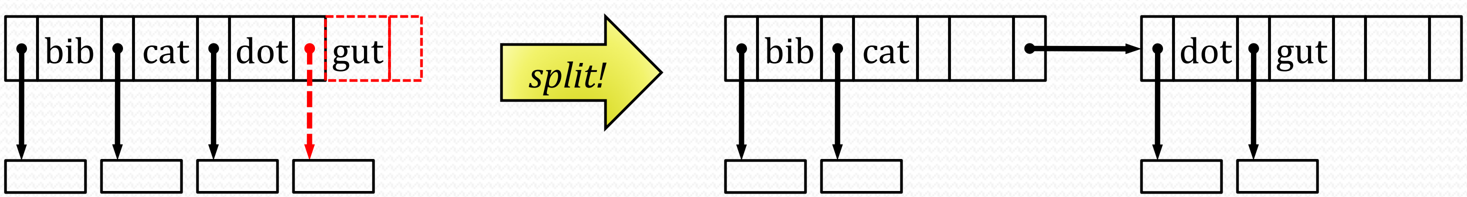


Splitting the Leaf-Node

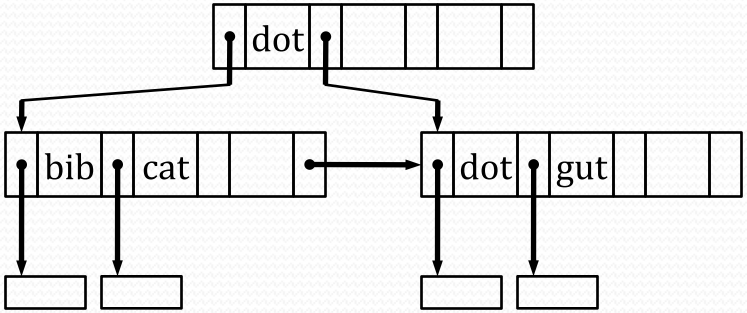
- If a leaf node overflows, must split it into two nodes!
- Our index after also inserting a “gut” record:



- Next we want to insert “dot”, but there isn’t room
  - Split the node into two nodes
  - Approx. half of the values remain in left node, and the rest are moved to the right node
  - The two leaf-nodes are chained together



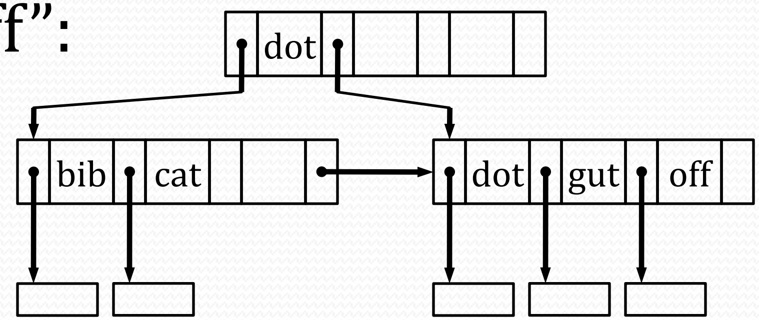
Splitting the Leaf-Node (2)



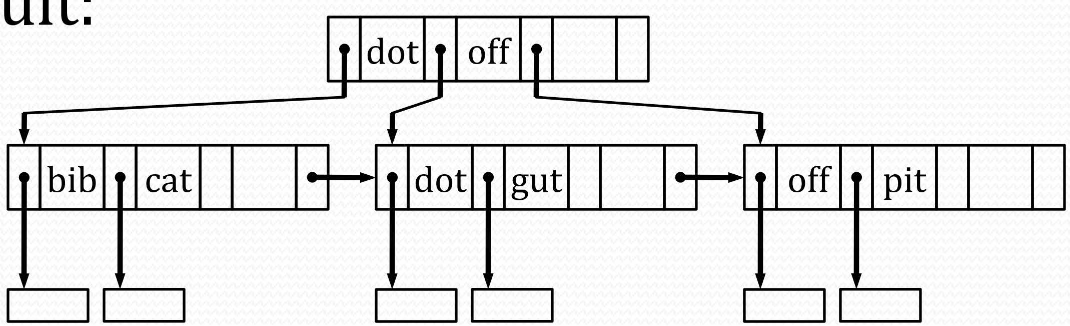
- We aren't done yet…
  - We need a new parent node to reference the two leaves
  - Will contain one key: "dot"
- General principle:
  - When a node is split into two, need to promote the new node's first key-value up to the parent-node's table
  - *<u>Note</u>: New node is always to right of the node being split*
- If there isn't a parent-node:
  - The root node is being split!
  - Create a new root node, and increase tree's depth by 1

Insertion Example, Cont.

- Our tree after also inserting "off":



- Now, want to insert "pit"
  - Again, split leaf node in two, and divide the leaf's values across the two nodes
  - Promote new node's first key-value to the parent
- Result:



B^+-Tree Insertion Algorithm

- Algorithm is generally straightforward to implement
- When splitting a leaf node, simplify process by using a temp memory area *T* that can hold overflowed node's contents
- Example: *L* is a full leaf-node
  - Want to add key *K* and associated record-pointer *P* to node *L*
- Implementation:
  - Copy contents of *L* into temporary memory block *T*
  - Insert new pair *K*, *P* into *T* *(it can hold the extra record)*
  - Create new empty leaf-node *L'*
  - Set $L'.P_n = L.P_n$, and set $L.P_n = L'$ *(chain leaves together)*
  - Clear *L*, and copy P_1, K_1 thru $P_{\lceil n/2 \rceil}$, $K_{\lceil n/2 \rceil}$ from *T* into *L*
  - Copy $P_{\lceil n/2 \rceil+1}$, $K_{\lceil n/2 \rceil+1}$ thru P_n, K_n from *T* into *L'*

B$^+$-Tree Insertion Algorithm

insert(value K, pointer P):
  if tree is empty:
    L = new empty leaf node
  else:
    L = find leaf where K should go,
      using earlier search algorithm
  if L has less than n – 1 keys:
    insert_in_leaf(L, K, P)
  else:
    split node L into L, L' using
      mechanism on prev. slide
    K' = smallest key in L'
    insert_in_parent(L, K', L')

insert_in_leaf(node L, value K, pointer P):
  if $K < L.K_1$:
    insert P, K into L before $L.P_1$
  else:
    find largest K_i in L less than K
    insert P, K into L after $L.K_i$

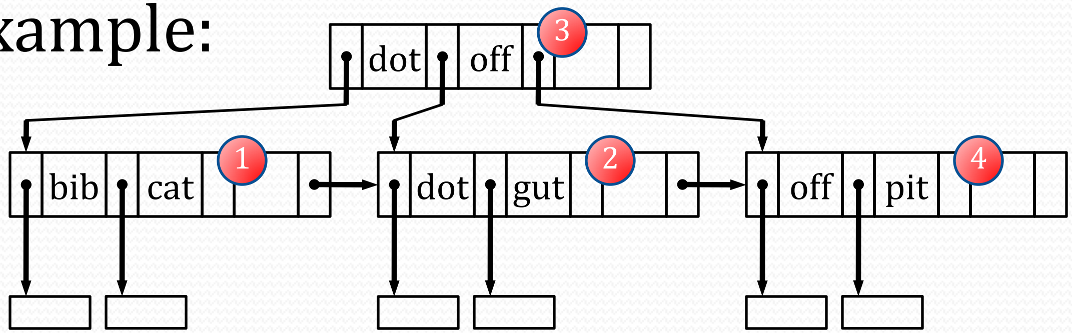
insert_in_parent(node N, value K', node N'):
  if N is root of tree:
    R = new empty non-leaf node
    set R contents to (N, K', N')
    make R the new root
  else:
    P = parent(N)
    if P has less than n pointers:
      insert (K', N') into P, just after N
    else:
      copy P to temporary block T
      insert (K', N') into T, just after N
      create new node P' ; clear P
      copy P_1, K_1 thru $P_{\lceil n/2 \rceil}$, $K_{\lceil n/2 \rceil}$ from T into P
      copy $P_{\lceil n/2 \rceil}$, $K_{\lceil n/2 \rceil}$ thru P_n, K_n from T into P'
      $K'' = P'.K_1$
      insert_in_parent(P, K'', P')

B^+-Tree Implementation Details

- Several additional details need to be maintained
  - e.g. type of node stored in each page (leaf/non-leaf/empty)
- Additionally, need to keep track of which node is B^+-tree's root node
  - As with table files, can store such details in page 0, and start the actual index pages with page 1
- Seems appealing to store additional structural details in B^+-tree nodes
  - The node's parent, siblings, etc.
  - Unfortunately, dramatically increases number of nodes that must be modified when manipulating the tree
  - Added complexity of using this simple structure is less costly than the additional IOs that would be required (!!!)

Implementation Details (2)

- Our simple B^+-tree example:



- Index file is still a linear sequence of pages
  - Pages in data file are in order of addition to the B^+-tree…
  - Over time, physical page order in data file will deviate widely from logical page order specified by the index
    - *(particularly the sequential traversal part)*
  - Periodically need to reorganize index pages to minimize number of disk seeks incurred by access/traversal