Outline

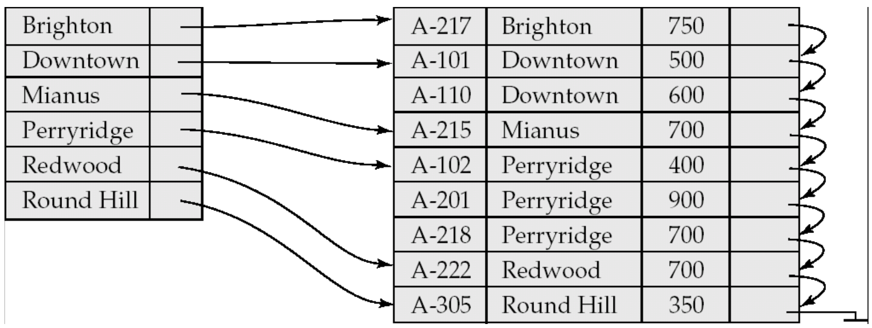
- Storage hierarchy
- Disks
- RAID
- Spark
- Buffer Manager
- File Organization
- Indexes
- B+-Tree Indexes
- Etc..

Index

- A data structure for efficient search through large databases
- Two key ideas:
  - The records are mapped to the disk blocks in specific ways
    - Sorted, or hash-based
  - Auxiliary data structures are maintained that allow quick search
    - Think library index/catalogue
- *Search key*:
  - Attribute or set of attributes used to look up records in indexes
  - E.g. SSN for a persons table
  - *Can be different from candidate or primary keys*
- Two main types of indexes
  - Ordered indexes
  - Hash-based indexes

Ordered Indexes

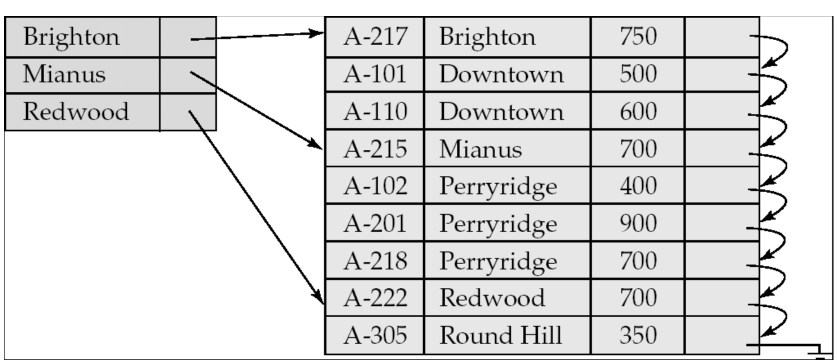
- We assume ordered indexes are sorted by search key
- Primary (“clustered”) indexes
  - File ordering = search key
  - Can have only one primary index on a relation
- Secondary (“nonclustered) index
  - File ordering != search key



- *dense* means every search value has an index entry

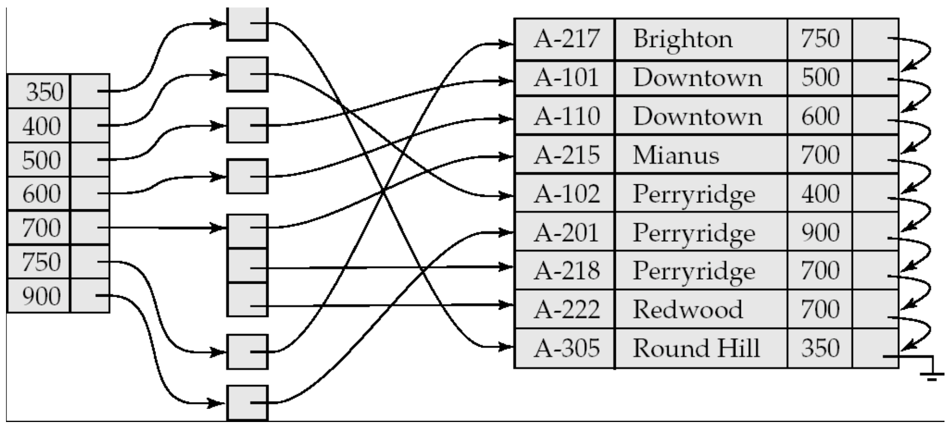
Primary *Sparse* Index

- Index doesn’t need every key
  - Allows for very small indexes
  - Better chance of fitting in memory
  - Tradeoffs?
    - Some amount of in-memory search
    - Must access the relation file even if the record is not present



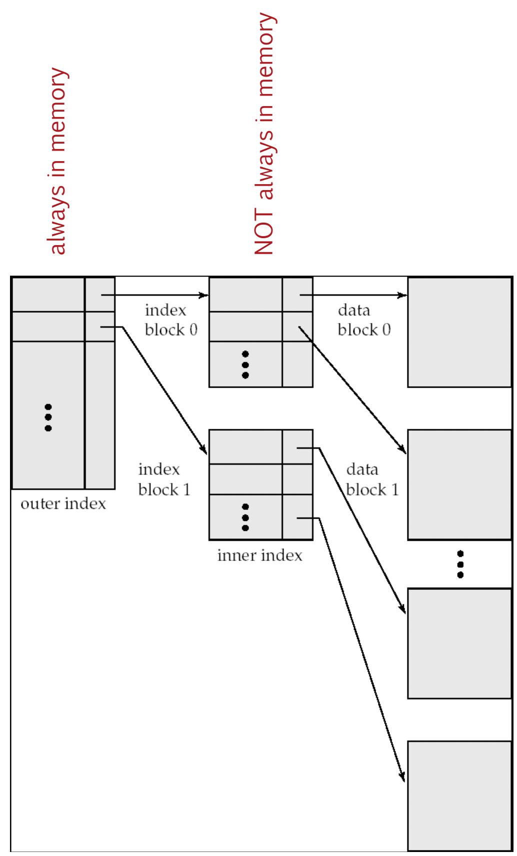
Secondary Index

- Relation sorted on *branch* (not search key)
- But we want an index on *balance*
- Must be dense



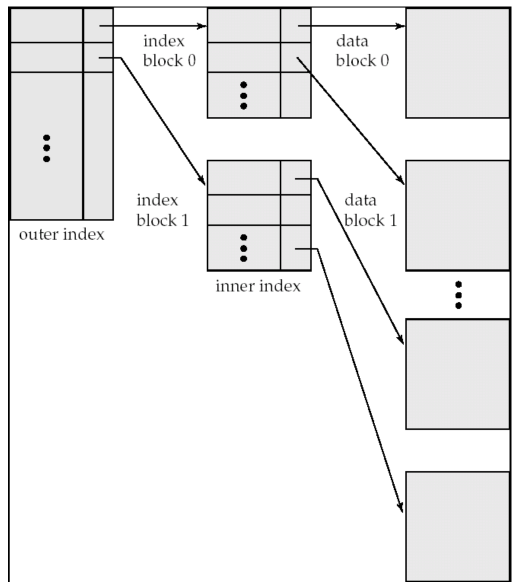
Multi-level Indexes

- What if the index itself is too big for memory?
- Assume:
  - relation size = 1,000,000,000:
  - block size = 100 tuples
- Then:
  - number of pages = 10,000,000
  - 16 bytes/entry is 120 MB for index
  - This is too much…
- Solution
  - Build an index on the index itself



Multi-level Indexes

- How do you search through a multi-level index ?
- Same search keys



Multi-level Indexes

- What about keeping the index up-to-date ?
  - Tuple insertions and deletions
    - Need to modify index as data is modified
    - This is a static structure
    - Need overflow pages to deal with insertions
  - Works well if no inserts/deletes
  - Not so good when inserts and deletes are common

Hash Indexes

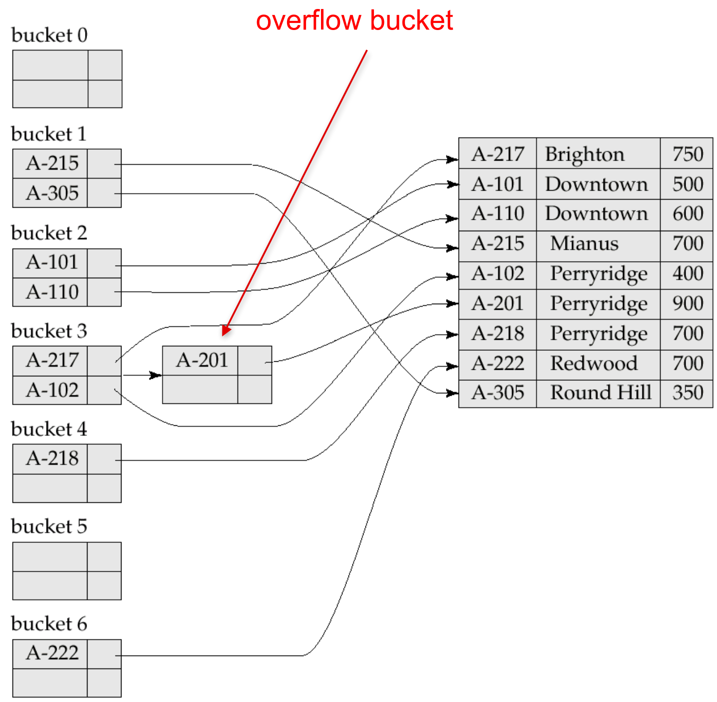
Extends the basic idea

Search:
  Find the bucket with search key
  Search the bucket
  Follow the pointer

Range search ?
  a < X < b ?

Must be dense.

Often used for secondary indexes.



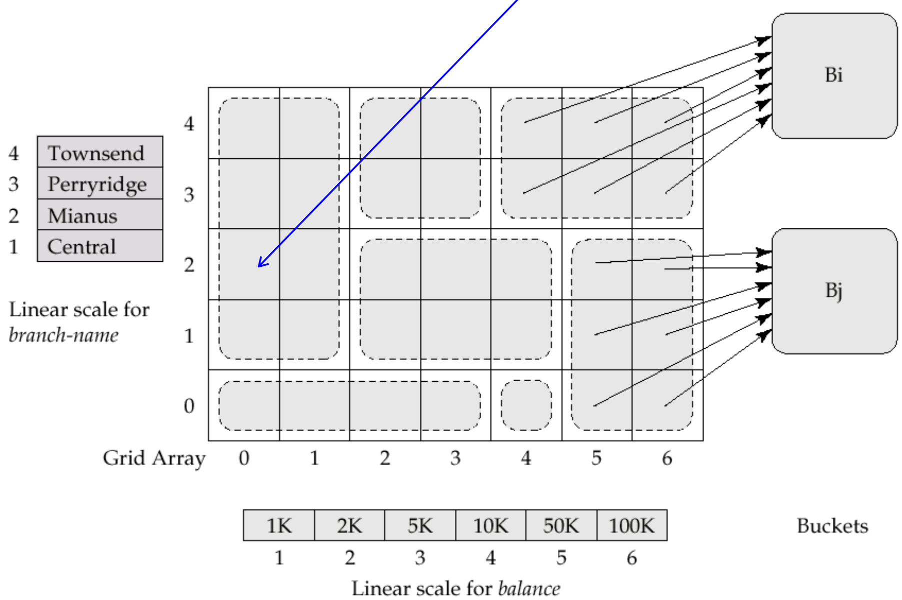
Hash Indexes

- Very fast search on equality
- Can't do range searches at all
  - Must scan the file
- Inserts/Deletes
  - Overflow pages can degrade the performance
  - Two approaches
    - Dynamic hashing (rehashing using new hash alg)
    - Extendible hashing (rehashing using more hash bits (trie))
      - bucket at a time can be extended (rehashed w/ more bits)

Grid Files

Multidimensional index structure
Can handle range queries:
branch = "pox" and balance = 2002
branch >= "Central" and balance < 5000

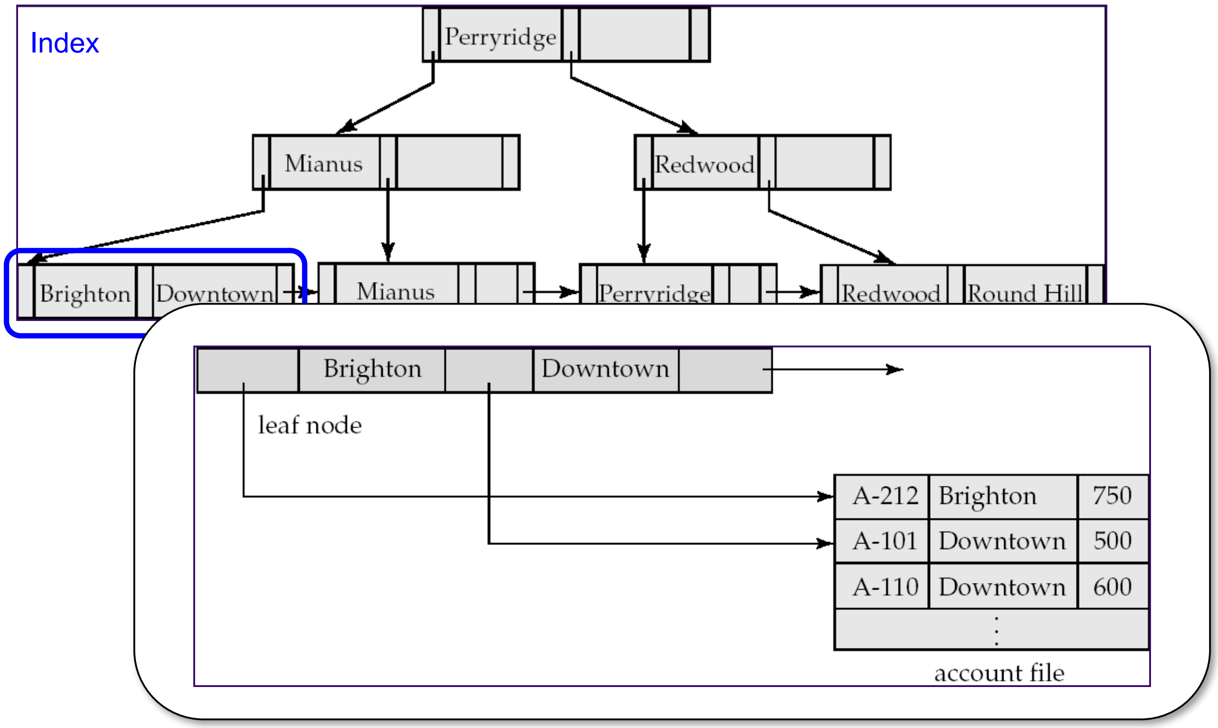
Stores pointers to tuples with :
branch between Mianus and Perryridge
and balance < 1k



Outline

- Storage hierarchy
- Disks
- RAID
- Spark
- Buffer Manager
- File Organization
- Indexes
- B+-Tree Indexes
- Etc..

Example B+-Tree Index



B+-Tree Node Structure

- Typical node

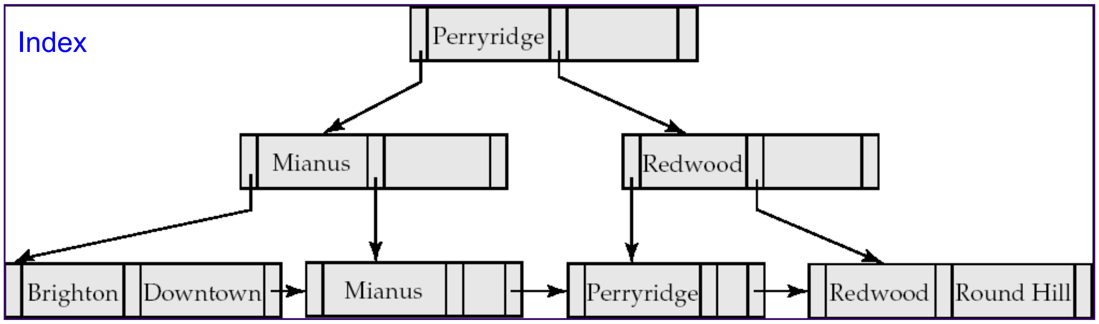
P_1	K_1	P_2	...	P_{n-1}	K_{n-1}	P_n

  - K_i are the search-key values
  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$

- Height is *#edges from leaf to root*

Example B+-Tree Index



- $n = 3$ (at most 3 ptrs)
  - each interior node can have up to n children
  - each leaf can have up to $n-1$ keys
- $h = 2$ (height)

Properties of B+-Trees

- It is balanced
  - Every path from the root to a leaf is same length
- Leaf nodes (at the bottom)
  - P_1 contains the pointers to tuple(s) with key K_1
  - …
  - P_n is a pointer to the *next* leaf node
  - Up to n-1 key values
  - Must contain at least $\lceil \frac{n-1}{2} \rceil$ key values
  - n=4 implies at most 4 pointers, up to 3 values, minimum of 2 values

P_1	K_1	P_2	$\ldots$	P_{n-1}	K_{n-1}	P_n

Properties

- Interior nodes

P_1	K_1	P_2	$\ldots$	P_{n-1}	K_{n-1}	P_n

  - All tuples in the subtree pointed to by P_i, have search key $< K_i$
  - To find a tuple with key $K_{j'} < K_i$, follow P_i
  - Contains:
    - at most n pointers
    - at least $\lceil \frac{n}{2} \rceil$ pointers (unless root)