CS 133: Databases

Fall 2019
Lec 04 – 09/12
Introduction to Indexes

Prof. Beth Trushkowsky

Administrivia

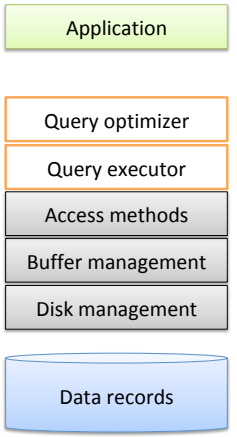
- Lab 1: Part 1 out and due next Wednesday
  - Submit your code on Gradescope
  - This first part graded for effort (P/F)… no slip days
  - Lab 1 will be graded in its entirety when complete
- Looking forward: no class next Thursday
  - Shorter problem set released that day

Goals for Today

- Understand the structure of tree-based indexes typically used in relational DBMSs
- See how indexes relate to file organization
- Reason about the cost tradeoffs for indexes

Simplified RDBMS Architecture

Heap files,
sorted files,
indexes



Given a query, the optimizer can **choose between available access paths** for each relation

Sequential scan and indexes!

Recall: Heap vs. Sorted Files

- Heap File
  - Long search time
  - Easy to maintain
- Sorted File
  - Use binary search (I/Os from disk)
  - Hard to maintain

if Students records sorted by ***name***, how to search for a particular ***gpa***?

What is the tradeoff of keeping pages ***not*** as packed?

Indexes to the Rescue

- Index: a **disk-based** data structure that speeds up selections on some ***search key fields***
  - Any subset of the fields of a relation can be the search key for an index on the relation
  - **Note**: search key not necessarily same as candidate (or primary) key
- Can have multiple indexes on the same relation
  - E.g., index on **Students.*sid*** and index on **Students.*name***

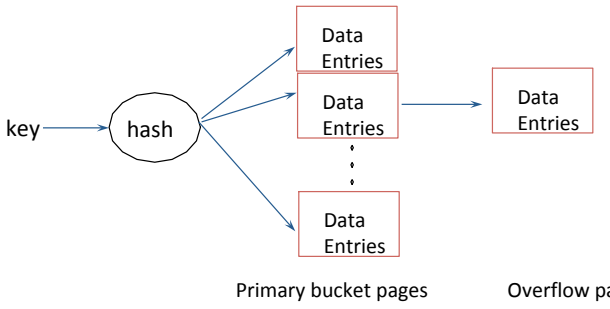
Also, can have *composite* indexes that involve multiple fields

Indexes: Overview

- An index
  - Contains a collection of ***data entries***
  - Supports efficient retrieval of all data entries with a given search key value **k**
    - textbook refers to these data entries as **k***
  - **Also may contain auxiliary information** that directs searches to the desired data entries
- Many **indexing techniques** exist
  - Tree-based, hash-based, R-trees…
  - Not all support *range* search
- Database administrator (DBA) chooses indexes!

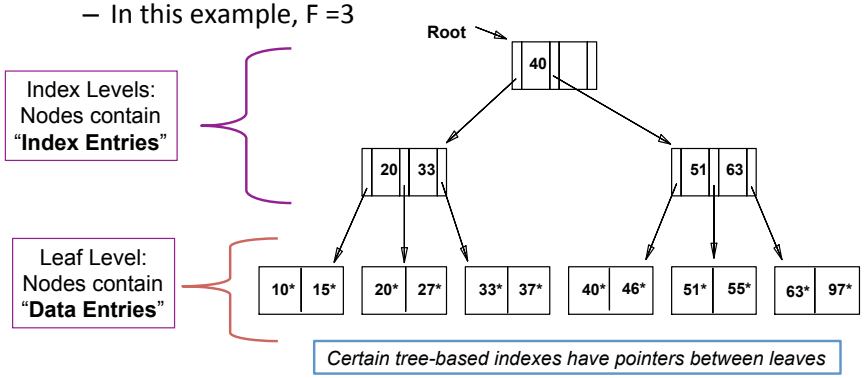
Anatomy of an Index: Hash Example

- Apply a hash function *h* to search key to determine which **data entries *bucket***



Anatomy of an Index: Tree Example

- Fast access to ***data entries*** at leaf level
- **Index entries**: <search key value, page id> direct search for **data entries**
- **Fan-out** (F): avg # children for non-leaf node
  - In this example, F =3



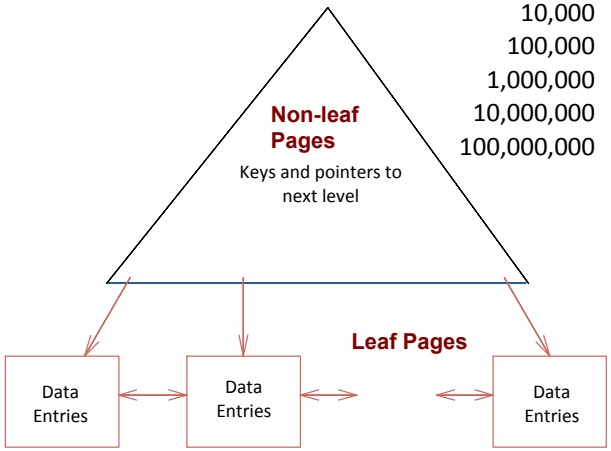
Exercise 2

How many levels will there be in a tree if there are B leaf pages and a fan-out of F?

$\log_F B + 1$

Typical Tree Fan-out

# Leaf Pages	Fanout	Levels
1,000	300	3
10,000	300	3
100,000	300	4
1,000,000	300	4
10,000,000	300	4
100,000,000	300	5



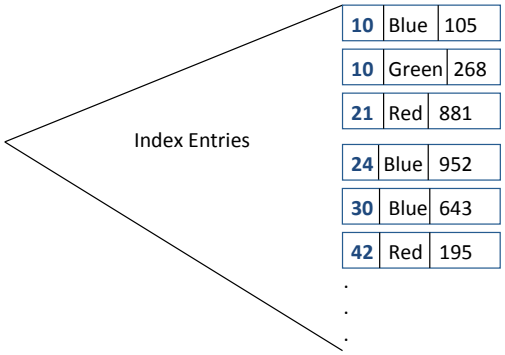
Data Entries

- What is in a **data entry** k* ?
  Recall, data entries live on:
  - The pages that are leaves of a tree-based index
  - The pages that correspond to buckets for hash index
- Three ***alternatives*** for data entry:
  1. Actual data record with search key value **k**
  2. <**k**, record id of matching record> pair
  3. <**k**, record ids of matching records> pair

Alternative choice is independent of choice of indexing technique (e.g., tree vs. hash)!

Alternative 1: k* is a data record

Employee {age, favorite_color, eid}
(tree index on *age*)



Alternative 1: Index-Organized File

- Actual data records stored in the data entries
  - E.g., leaves of tree or buckets of the hash
- Implications
  - Index structure **becomes file organization** for relation
  - Can ***only have one such index for a given relation***
  - Less lookup time vs. Alternatives 2 and 3, but more expensive than them to maintain

Exercise 3(a)

I/O Operation Cost

B: The size of the data (in packed pages)

	Heap File	Sorted File (100% Occupancy)	Index-Organized File, Tree (⅔ Occupancy)
Scan all records	B	B	**1.5 B** (because pages ⅔ full)
Equality Search (1 match)	0.5 B	$\log_2$ B	$\log_F 1.5B + 1$ (height + 1 reads)
Range Search	B	($\log_2$ B) + selectivity*B	$\log_F 1.5B$ + selectivity*1.5B
Insert (1 record)	2	($\log_2$B)+B (if read, write 0.5 file)	search +1 (1 to write, assume it fits)
Delete (1 match)	0.5B+1	($\log_2$B)+B (if read, write 0.5 file)	same as insert

Data Entries in Alternatives 2 and 3

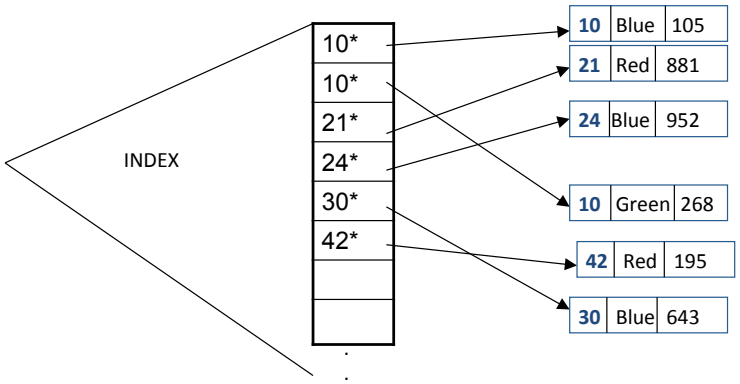
- Alternative 2
  {<k, record id of a matching data record>}

Act as pointers to where the data records are

- Alternative 3
  <k, {records ids of all matching data records}>
- Independent of the data file organization
  - If file has one index with alternative 1, all other indexes must be either alternative 2 or 3

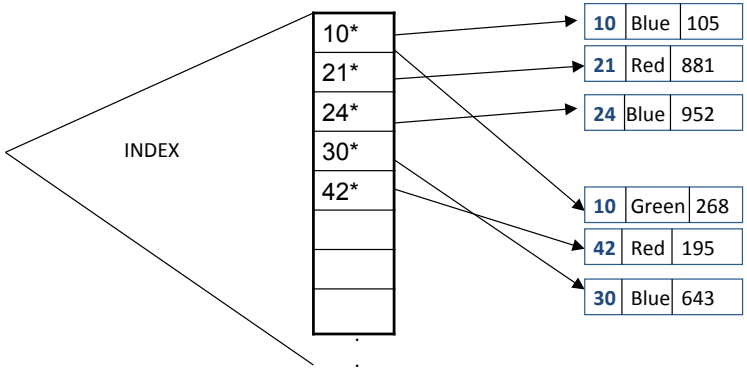
Alternative 2: One pointer per k*

Employee {age, favorite_color, eid}
(tree index on *age*)



Alternative 3: 1 or more pointers per k*

Employee {age, favorite_color, eid}
(tree index on *age*)

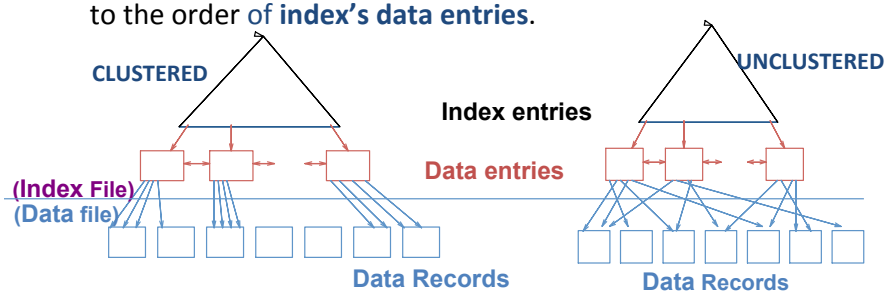


Clustered vs. Unclustered Index

- For Alts 2 or 3, we have ***two*** files – one holds the actual data records and one is for the index.
  - Order of index's data entries may or may not be same as data records

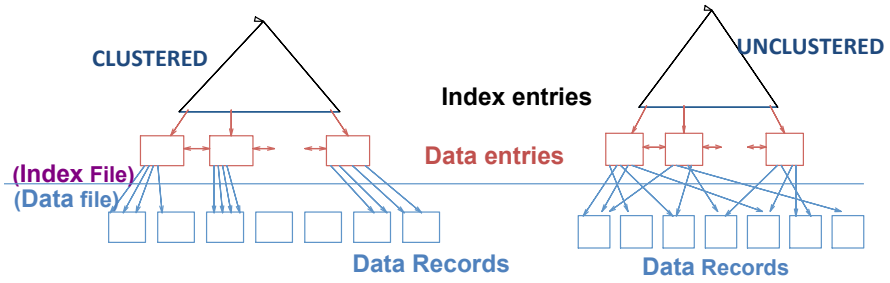
What about data entries order for Alternative 1?

- **Clustered index**: order of **data records** is same as or close to the order of **index's data entries**.



Alt 2: Clustered vs. Unclustered

- Clustered **Pros**
  - More efficient for range searches: scan leaves of tree
- Clustered **Cons**
  - Maintenance cost (pay on the fly or be lazy with reorganization)
  - One clustered index per data file (sorted on one search key)



Exercise 3(b) and (c)

I/O Operation Cost

B: The size of the data (in pages)

	Unclustered Alt-2 Tree Index (Index file: ⅔ = 67% occupancy) (Data file: 100% occupancy)	**Clustered** Alt-2 Tree Index (Index and Data files: ⅔ = 67% occupancy)
Scan all records	**B** ***(ignore index… why?)***	**1.5 B** ***(ignore index)***
Equality Search	**$(\log_F 0.5B) + 1 + 1$** *assume an index or data entry is 1/3 size of a record, so # pages at leaf level = .33 * 1.5B = 0.5B*	**$(\log_F 0.5B) + 1 + 1$**
Range Search	**$(\log_F 0.5B)$** **+ selectivity*0.5B** **+ selectivity*N**	**$(\log_F 0.5B)$** **+ 1** **+ selectivity*1.5B**
Insert	**search + 1 + 2** (2 for r/w in heap)	**search + 1 + 2**
Delete	**same as insert**	**same as insert**