CAS CS 460/660
Introduction to Database Systems

File Organization and Indexing

Slides from UC Berkeley

Review: Files, Pages, Records

- Abstraction of stored data is “files” of “records”.
  - Records live on *pages*
  - Physical Record ID (RID) = <page#, slot#>
- *Variable length* data requires more sophisticated structures for records and pages. (why?)
  - Records: offset array in header
  - Pages: Slotted pages w/internal offsets & free space area
- Often best to be “lazy” about issues such as free space management, exact ordering, etc. (why?)
- Files can be unordered (heap), sorted, or kinda sorted (i.e., “clustered”) on a *search key.*
  - Tradeoffs are update/maintenance cost vs. speed of accesses via the search key.
  - Files can be clustered (or sorted) at most one way.
- Indexes can be used to speed up many kinds of accesses. (i.e., “access paths”)

Sorted Files

- Heap files are lazy on update - you end up paying on searches.
- Sorted files eagerly maintain the file on update.
  - The opposite choice in the trade-off
- Let's consider an extreme version
  - No gaps allowed, pages fully packed always
  - Q: How might you relax these assumptions?
- Assumptions for our BotE Analysis:
  - Files compacted after deletions.
  - Searches are on sort key field(s).

Average Case I/O Counts for Operations (B = # disk blocks in file)

	Heap File	Sorted File	Clustered File
Scan all records	B	B	
Equality Search (1 match)	0.5 B	$\log_2$ B (if on sort key) 0.5 B (otherwise)	
Range Search	B	($\log_2$ B) + selectivity * B	
Insert	2	($\log_2$B)+ B	
Delete	0.5B+1	Same cost as Insert	

The Problem(s) with Sorted Files

1) Expensive to maintain
   - Especially if you want to keep the records packed tightly.
   - Q: What if you are willing to relax that constraint?
2) Can only sort according to a single search key
   - File will effectively be a “heap” file for access via any other search key.
   - e.g., how to search for a particular student id in a file sorted by major?

Indexes: Introduction

- Sometimes, we want to retrieve records by specifying *values in one or more fields*, e.g.,
  - Find all students in the "CS" department
  - Find all students with a gpa > 3.0
  - Find all students in CS with a gpa > 3.0

- *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.
  - *Search key* is not the same as (primary) *key*
  - e.g., Search keys don't have to be unique.

Indexes: Overview

- An index contains a collection of *data entries*, and supports efficient retrieval of all records with a given search key value k.
  - Typically, index also contains auxiliary information that directs searches to the desired data entries (index entries)
- Many indexing techniques exist:
  - B+ trees, hash-based structures, R trees, …
- Can have multiple (different) indexes per file.
  - E.g. file sorted by *age*, with a hash index on *salary* and a B+tree index on *name*.

Index Classification

1. Selections (lookups) supported
2. Representation of data entries in index

   - what kind of info is the index actually storing?

   - we have 3 alternatives here
3. Clustered vs. Unclustered Indexes
4. Single Key vs. Composite Indexes
5. Tree-based, hash-based, other

Indexes: Selections supported

field <*op*> constant

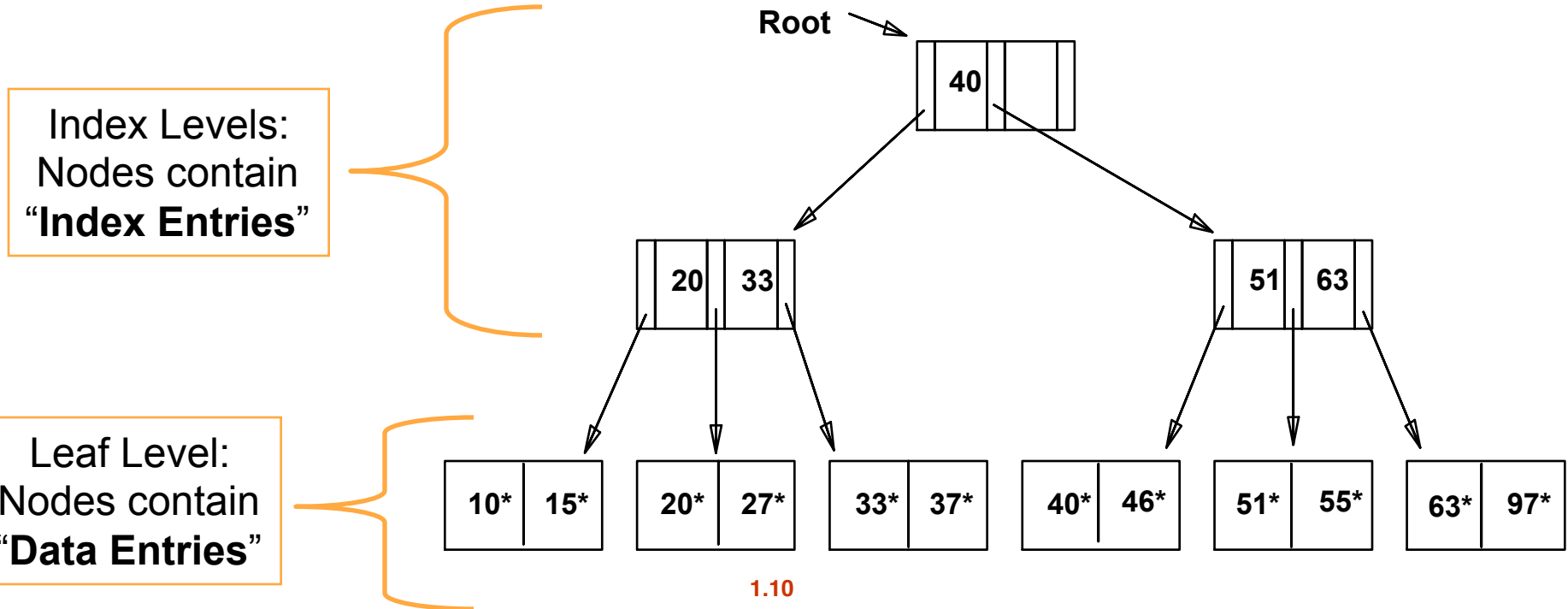
- Equality selections (*op* is =)
  - Either "tree" or "hash" indexes help here.
- Range selections (*op* is one of <, >, <=, >=, BETWEEN)
  - "Hash" indexes don't work for these.

More exotic selections

- multi-dimensional ranges ("between Brookline, Newton, Waltham, and Cambridge")
- multi-dimensional distances ("within 2 miles of Copley Sq")
- Ranking queries ("10 restaurants closest to Kenmore Sq")
- Regular expression matches, genome string matches, etc.
- Keyword/Web search - includes "importance" of words in documents, link structure, …

Tree Index: Example

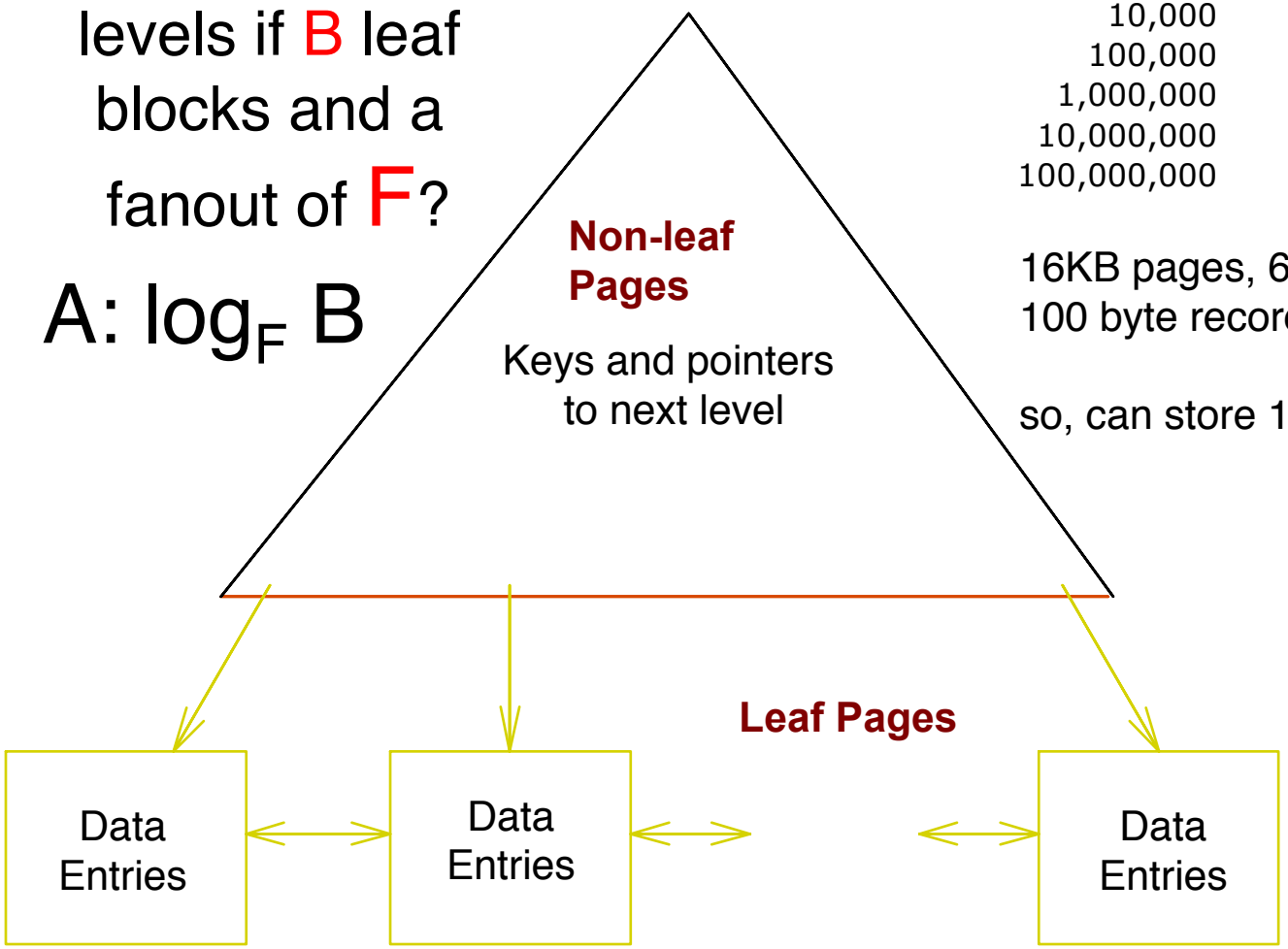
- *Index entries*:<search key value, page id>
  they direct search for data entries *in leaves.*
- In example: Fanout (F) = 3 (note: unrealistic!)
  - more typical: 16KB page, 67% full, 32Byte entries
  = approx 300



Index Fanout and Height

Q: How many levels if B leaf blocks and a fanout of F?

A: $\log_F B$



# Leaf Blocks	(Avg) Fanout	Levels
1,000	100	3
10,000	100	3
100,000	100	4
1,000,000	100	4
10,000,000	100	4
100,000,000	100	5

16KB pages, 67%full and
100 byte records = approx 100 recs/page.

so, can store 10B rows with 5 levels.

Note: All pages at all levels are: "Slotted Pages"

What's in a "Data Entry"?

- Question: What is stored in the leaves of the index for key value "k"? (a data entry for key "k" is denoted "k*" in book and examples)

- Three alternatives:
  1. Actual data record(s) with key value **k**
  2. {<**k**, rid of a matching data record>}
  3. <**k**, {rids of all matching data records}>

- Choice is orthogonal to the indexing technique.
  - e.g., B+ trees, hash-based structures, R trees, …

Alt 1= “Index-Organized File”

- Actual data records are stored in leaves.
  - If this is used, index structure becomes a file organization for data records (e.g., a sorted file).
  - At most one index on a given collection of data records can use Alternative 1.
  - This alternative saves pointer lookups but can be expensive to maintain with insertions and deletions.

Operation Cost

B: The size of the data (in pages)

	Heap File	Sorted File (100% Occupancy)	Tree Index-Organized File (67% Occupancy)
Scan all records	B	B	1.5 B (bcos 67% full)
Equality Search *unique key*	0.5 B	$\log_2 B$	$\log_F 1.5B$
Range Search	B	$(\log_2 B)$ + #match pg	$(\log_F 1.5B)$ + #match pg
Insert	2	$(\log_2 B)+B$	$(\log_F 1.5B)+1$
Delete	0.5B+1	$(\log_2 B)+B$ *(because rd,wrt 0.5 file)*	$(\log_F 1.5B)+1$

RIDs in Data Entries

Alternative 2

{<k, rid of a matching data record>}

and Alternative 3

<k, {rids of all matching data records}>

- Easier to maintain than Index-Organized.
  - but: Index-organized could be faster for reads.
- For a given file, at most one index can use Alt 1 (index organized); rest must use 2 or 3.
- Alt 3 more compact than Alt 2, but:
  - Has *variable sized* data entries
  - For large rid lists could span multiple blocks!

Clustered vs. Unclustered Index

"Clustered" Index: order of data records is same as or `close to' the order of index data entries.

A file can be clustered on at most 1 search key.

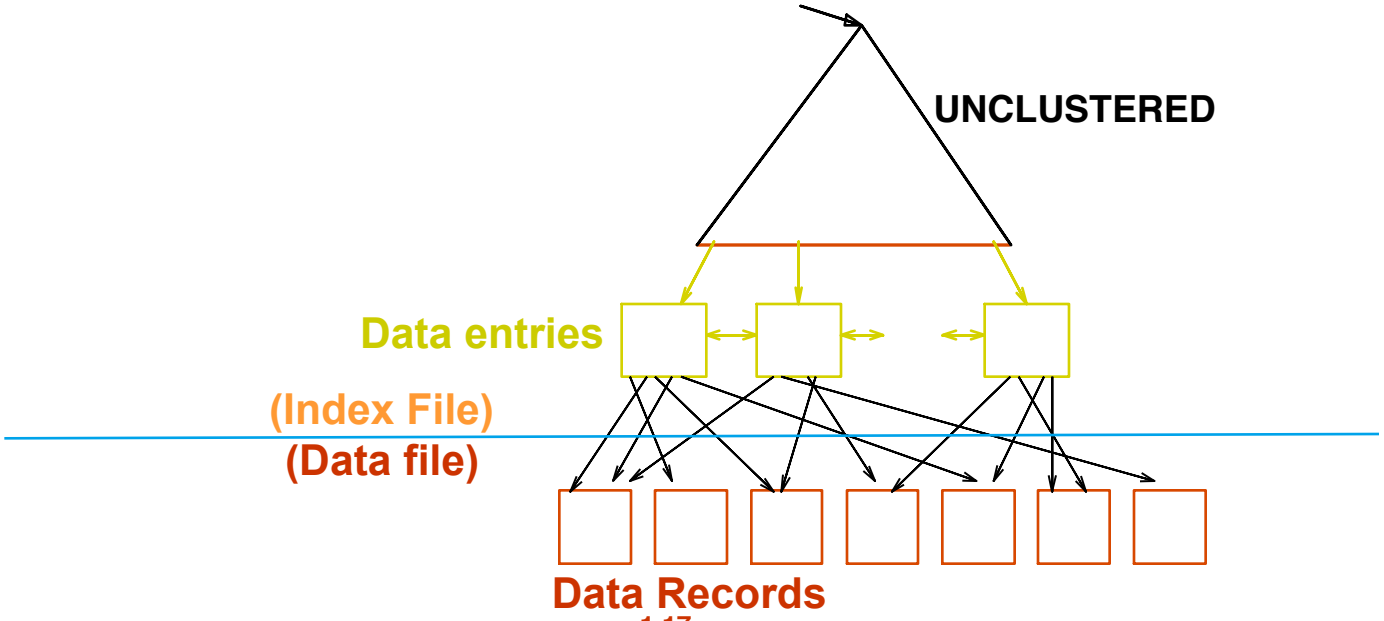
Cost of retrieving data records via index varies *greatly* based on whether it is clustered or not!

- Index-organized implies clustered *but not vice-versa*.
  - In other words, alt-1 is always clustered
  - alt 2 and alt 3 may or may not be clustered.

Ex: Alt 2 index for a Heap File

For alts 2 or 3, we typically have two files – one for data records and one for the index.

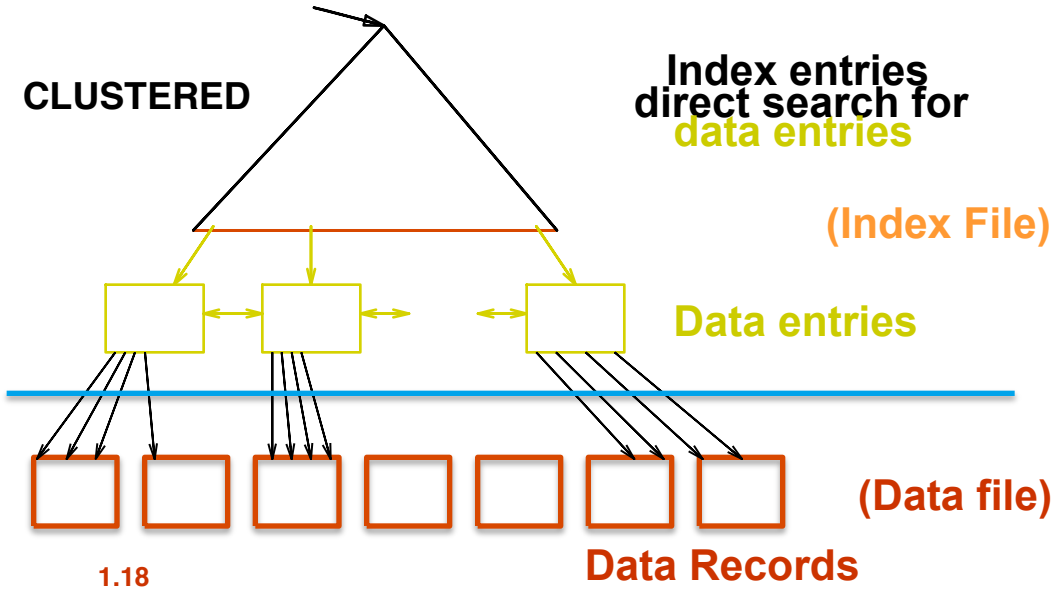
For an unclustered index, the order of data records in the data file is unrelated to the order of the data entries in the leaf level of the index.



Ex: Alt 2 index for a Heap File

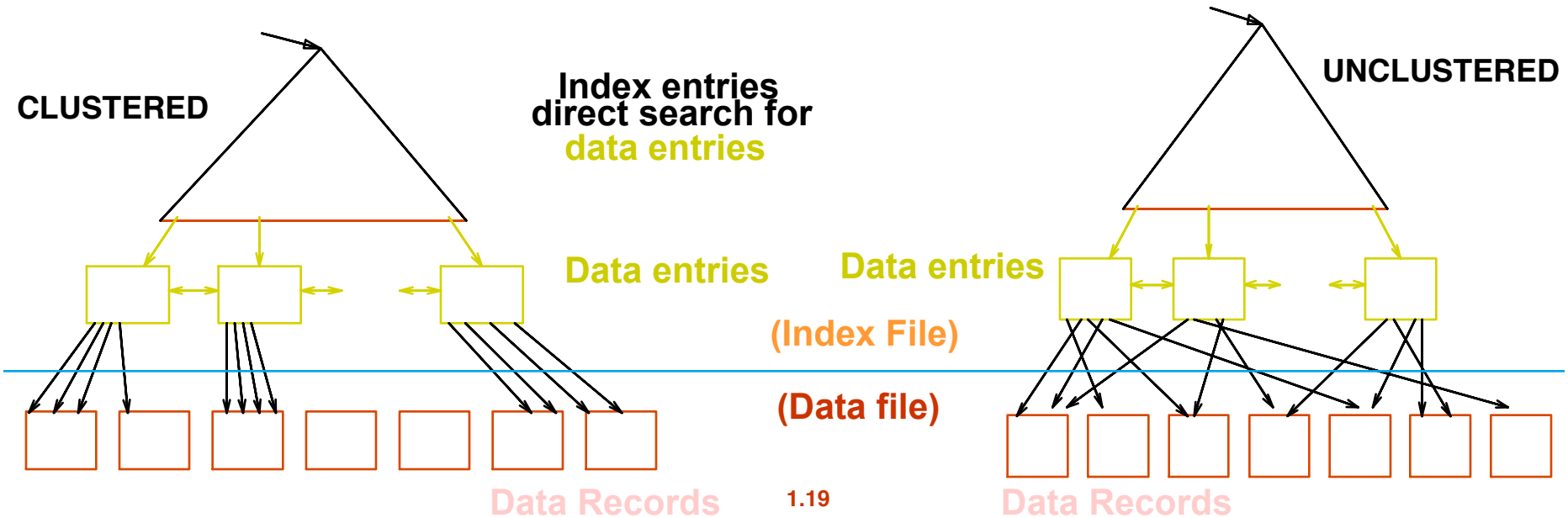
For a clustered index:

- Sort the heap file on the search key column(s)
  - Leave some free space on pages for future inserts
- Build the index
- Use overflow pages in data file if necessary
  - Thus, clustering is only approximate – data records may not be exactly in sort order (can clean up later)



Clustered vs. Unclustered

- Clustered Pros
  - More efficient for range searches
  - May be able to do some types of compression
- Clustered Cons
  - Maintenance cost (pay on the fly or be lazy with reorganization)
  - Can only cluster according to a single search key



Operation Cost

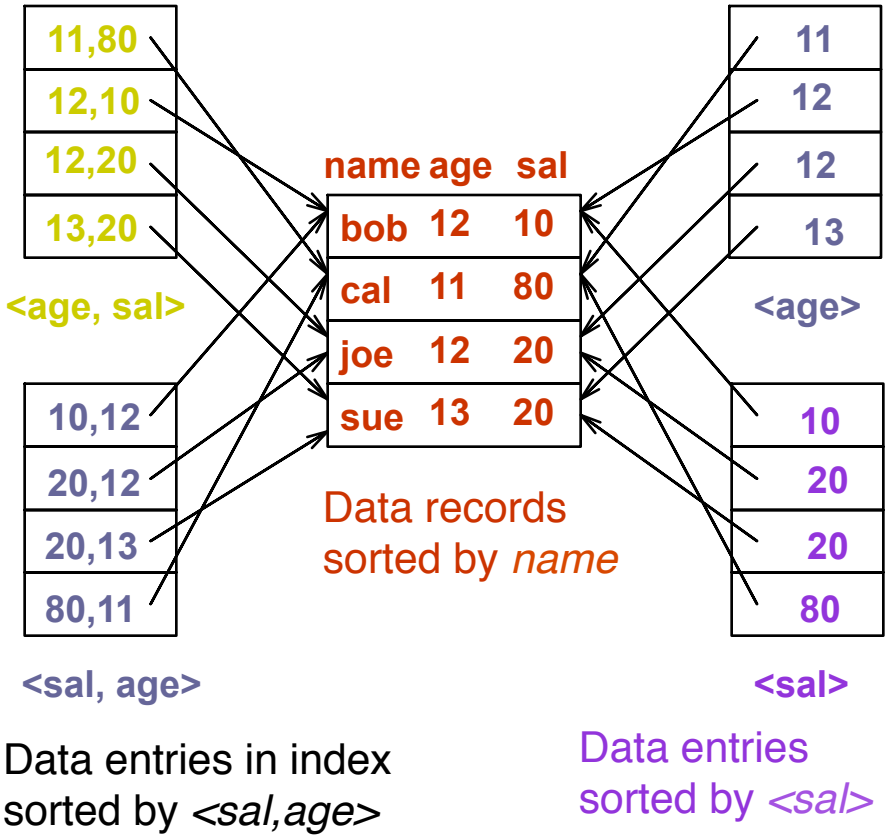
B: The size of the data (in pages)

	Unclustered Alt-2 Tree Idx (Index file: 67% occupancy) (Data file: 100% occupancy)	Clustered Alt-2 Tree Index (Index and Data files: 67% occupancy)
Scan all records	B (ignore index)	1.5 B (ignore index)
Equality Search *unique key*	$1 + \log_F 0.5\,B$ assume an index entry is 1/3 the size of a record so index leaf level = .33 * 1.5B = 0.5B	$1 + \log_F 0.5B$
Range Search	$(\log_F 0.5B)$ + #matching_leaf_pages + #match records	$(\log_F 0.5B)$ + #match_leaf_pgs + #match pages
Insert	$(\log_F 0.5B)+3$	$(\log_F 0.5B)+3$
Delete	same as insert	same as insert

Composite Search Keys

- Search on a combination of fields.
  - Equality query: Every field value is equal to a constant value. E.g. wrt <age,sal> index:
    - age=20 and sal =75
  - Range query: Some field value is not a constant. E.g.:
    - age > 20; or age=20 and sal > 10
- Data entries in index sorted by search key to support range queries.
  - Lexicographic order
  - Like the dictionary, but on fields, not letters!

Examples of composite key indexes using lexicographic order.



Index Classification Revisited

1. Selections (lookups) supported
2. Representation of data entries in index
   - what kind of info is the index actually storing?
   - 3 alternatives here
3. Clustered vs. Unclustered Indexes
4. Single Key vs. Composite Indexes
5. Tree-based, hash-based, other