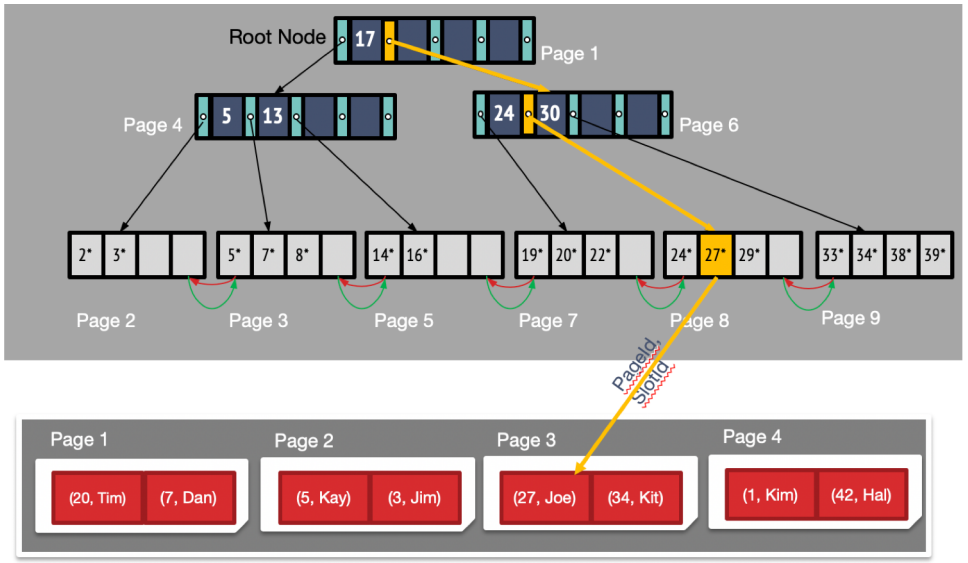
Index Files

Alvin Cheung
Fall 2022

Reading: R & G Chs 9-10

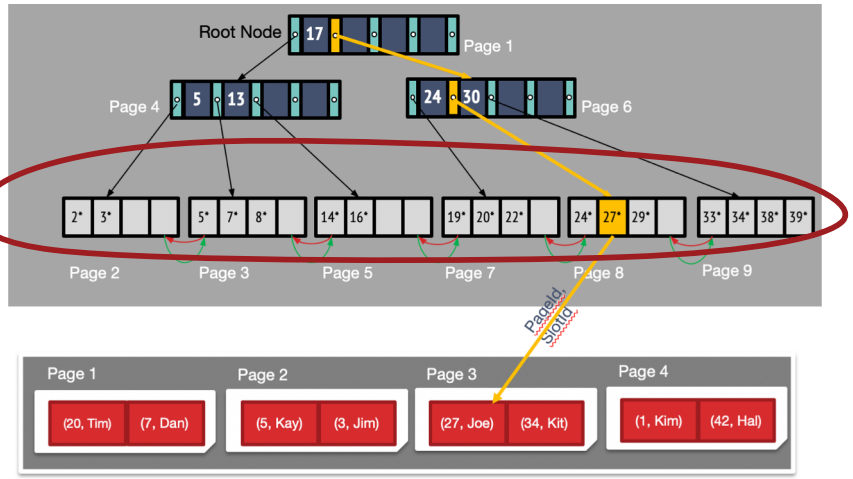
Connecting Back to the Storage Layer



- We talked about a B+tree pointing to unordered pages in a heap file
- This is not the only approach

- Let's talk about various alternatives for the B+ tree's:
  - Leaf nodes (the interface between index and the data)
  - Heap file (the actual data)

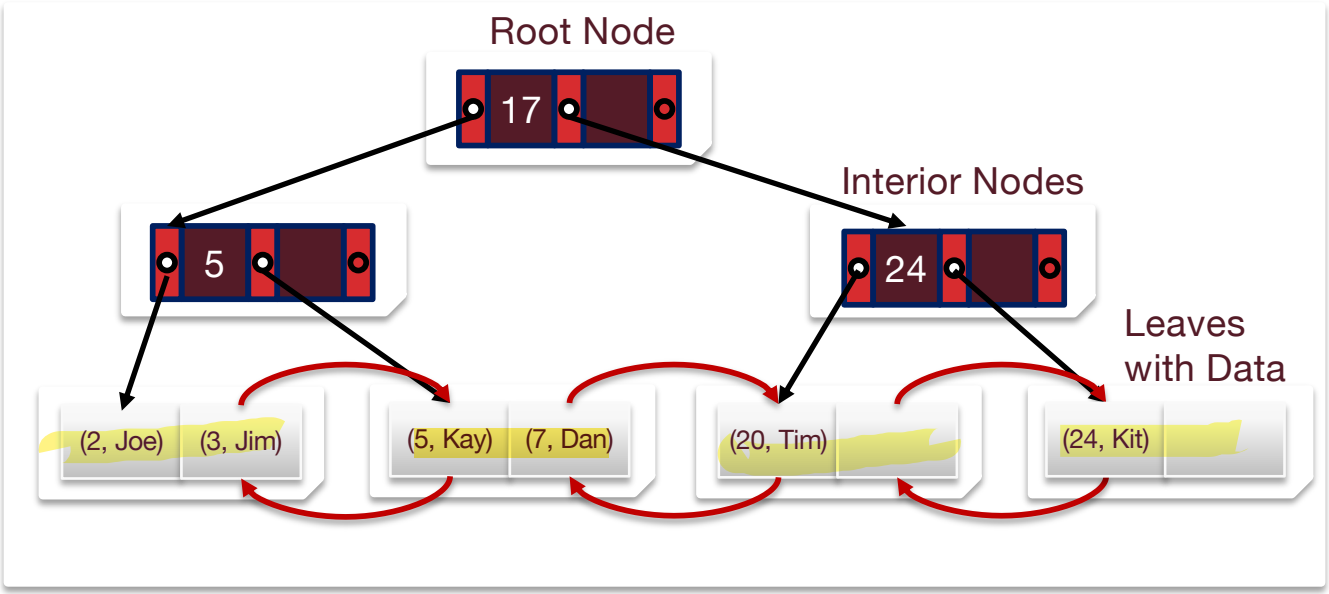
Three basic alternatives for leaf nodes



- Also applies for data entries for other types of indexes besides B+ trees
- Three basic alternatives (Textbook uses same numbering!)
  - Alternative 1: By Value
  - Alternative 2: By Reference ← *we saw this*
  - Alternative 3: By List of references

Alternative 1: By Value

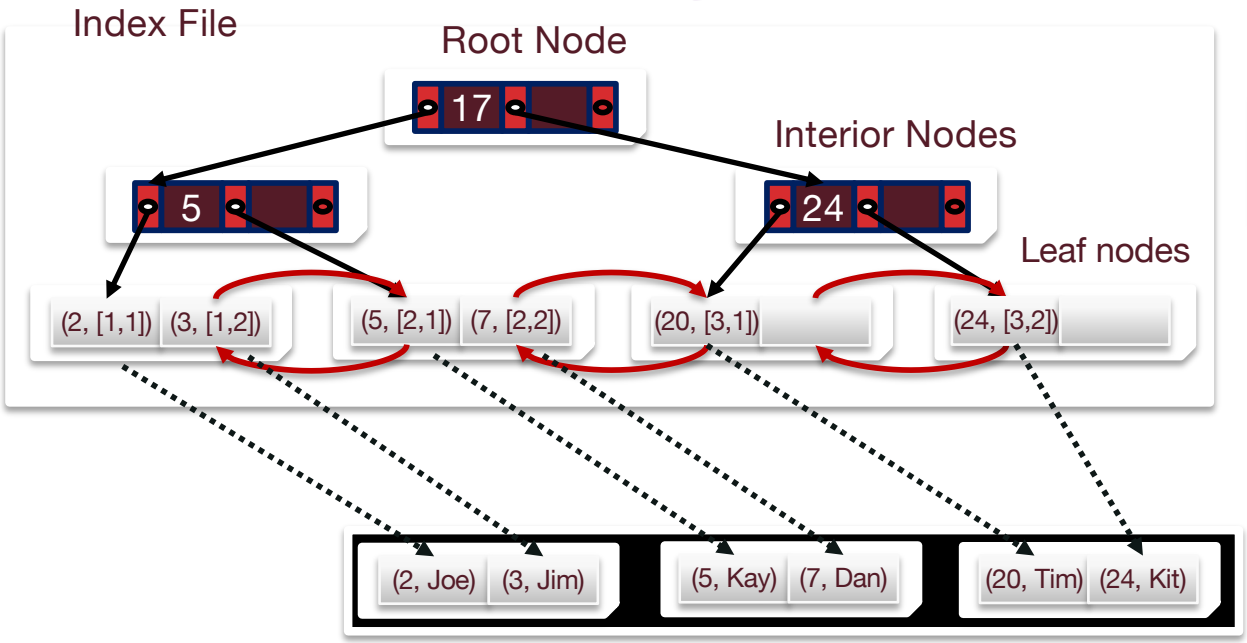
- Leaf pages store records directly
  - No need to follow pointers



uid	name
2	Joe
3	Jim
5	Kay
7	Dan
20	Tim
24	Kit

Alternative 2: By Reference Pairs

- For each **k**, store recordId of matching data record as pairs
  - Each entry in leaf: <k, recordId>
    - Recordid = [page id, slot id]
  - We used this previously

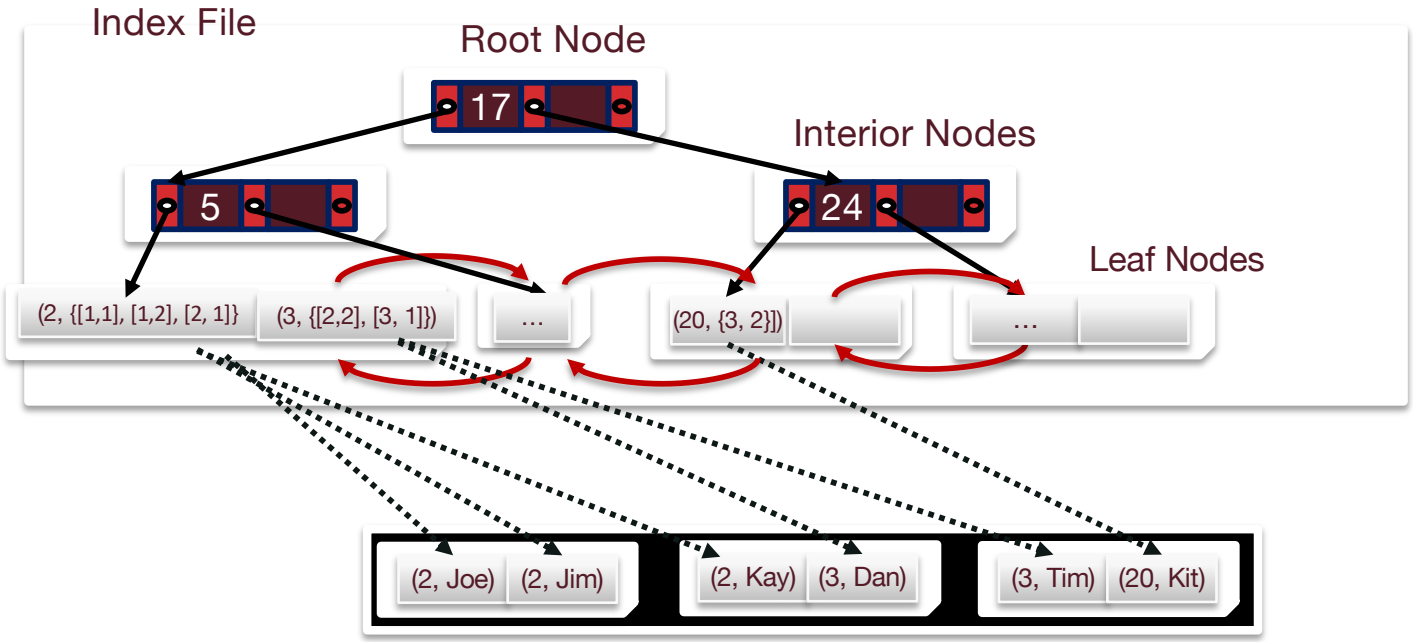


Index Contains
(Key, Record Id)
Pairs

uid	name
2	Joe
3	Jim
5	Kay
7	Dan
20	Tim
24	Kit

Alternative 3: By Reference List

- For each k, store recordIds of matching records as a list
  - Each leaf entry: <**k**, {list of rids of matching data records}>
  - Alternative 3 more compact than alternative 2 (smaller tree)
    - Very large rid lists can span multiple blocks, needs bookkeeping to manage that
    - Can handle non-unique search keys



Index Contains
(Key, {list of record Id}) Pairs

By Value vs. By Reference

- Both Alternative 2 and Alternative 3 index data *by reference*

multiple B+ trees for same table

- If we want to support multiple indexes per table, by reference is *required*
  - Otherwise we would be replicating entire tuples
  - Q: Why is replicating a problem?
  - Replicating data leads to complexity during updates, so we want to avoid
    - Need to make sure that all copies of the data are kept in sync.

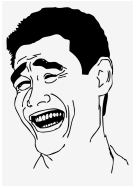
Connecting Back to the Storage Layer

- We talked about a B+tree pointing to unordered pages in a heap file
- This is not the only approach

- We'll talk about various alternatives for the:
  - Leaf nodes (the interface between index and the data)
  - Heap file (the actual data, if outside the index)  *this is next*

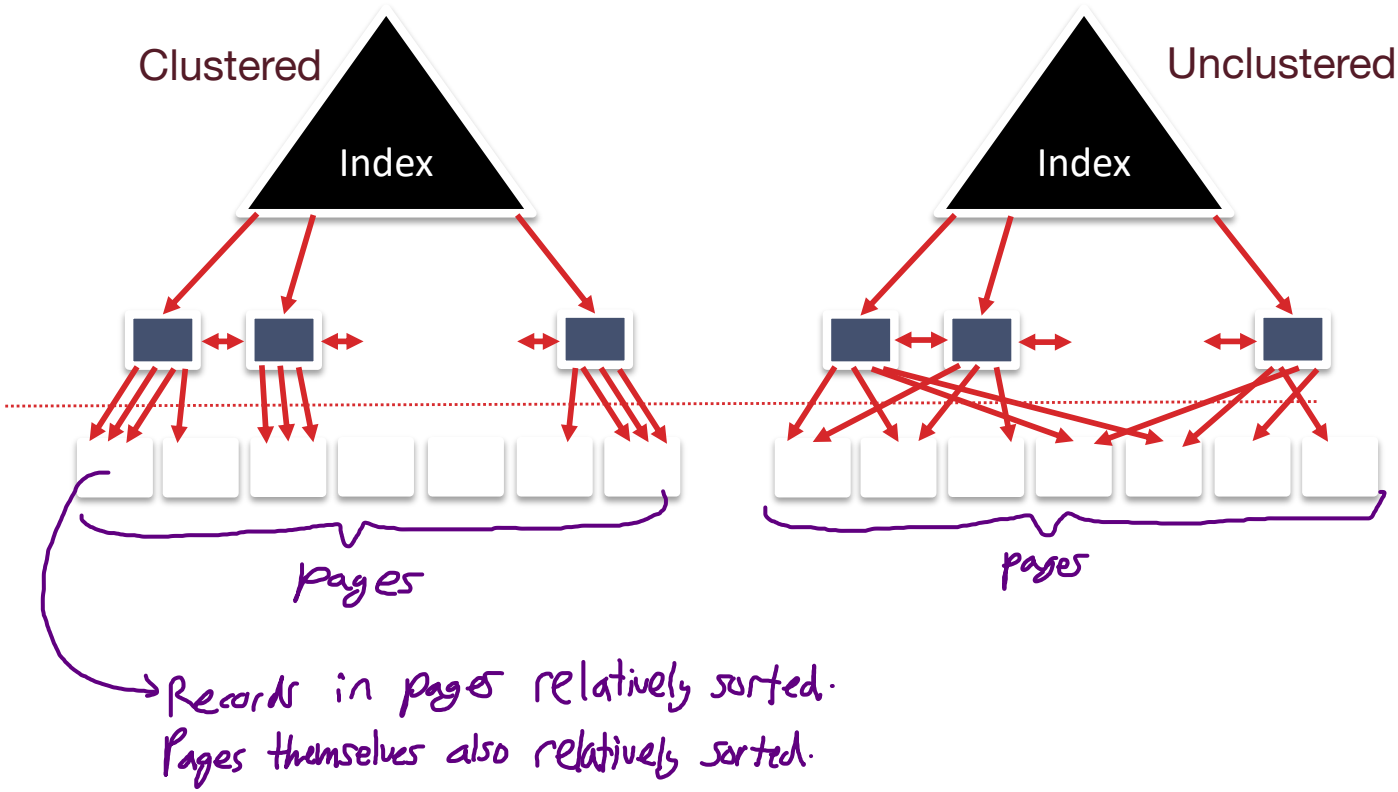
Clustered vs. Unclustered Index

- By-reference indexes (Alt 2 and 3) can be *clustered* or *unclustered*
  - In reality, this is a property of the heap file associated with the index!
- Clustered index: Records relative sorted in page
  - Heap file records are kept *mostly* ordered according to **search keys** in index
    - Heap file order need not be perfect: this is just a performance hint
    - As we will see, cost of retrieving data records through index varies greatly based on whether index is clustered or not!
- Note: nothing to do with “clustering” in AI / machine learning:
  - grouping nearby items in a high-dimensional space or network



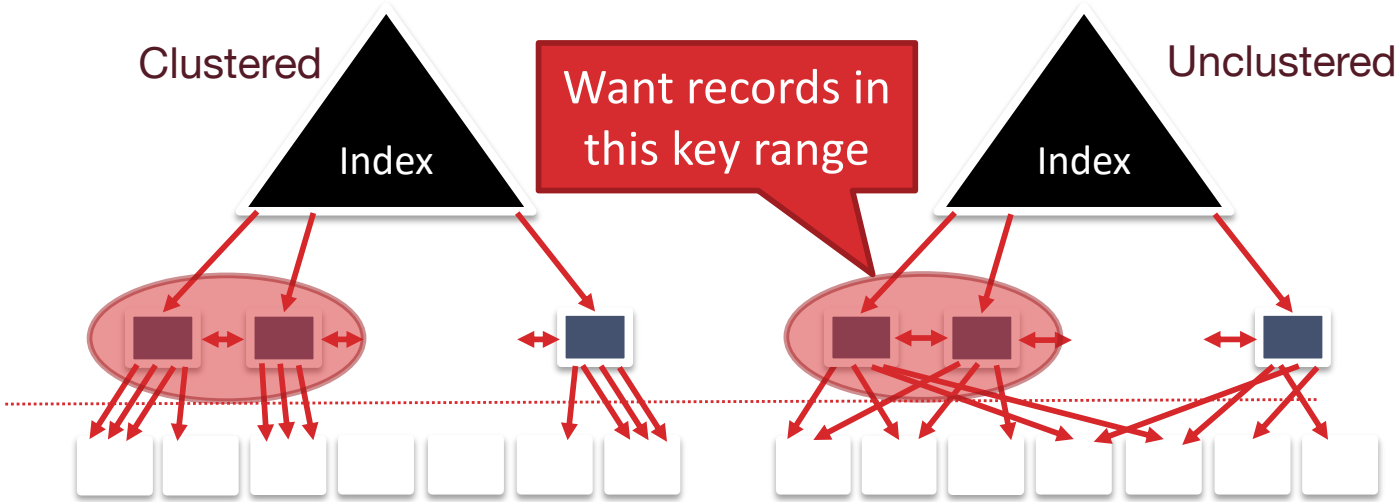
Clustered vs. Unclustered Index Visualization 1

- To build a clustered index, first sort the heap file
  - Leave some free space on each block for future inserts
  - We then try to respect this order "as much as possible"
- In an unclustered index, there is no such restriction



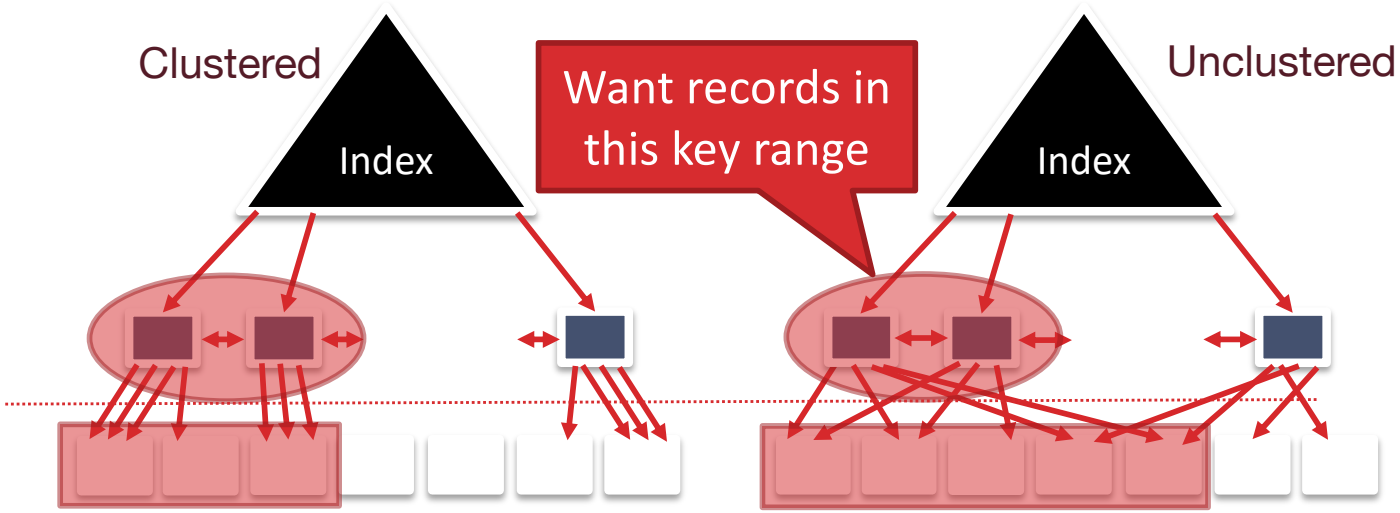
Clustered vs. Unclustered Index Visualization 2

- To build a clustered index, first sort the heap file
  - Leave some free space on each block for future inserts
  - We then try to respect this order “as much as possible”
- In an unclustered index, there is no such restriction



Clustered vs. Unclustered Index Visualization 3

- To build a clustered index, first sort the heap file
  - Leave some free space on each block for future inserts
  - We then try to respect this order "as much as possible"
- In an unclustered index, there is no such restriction

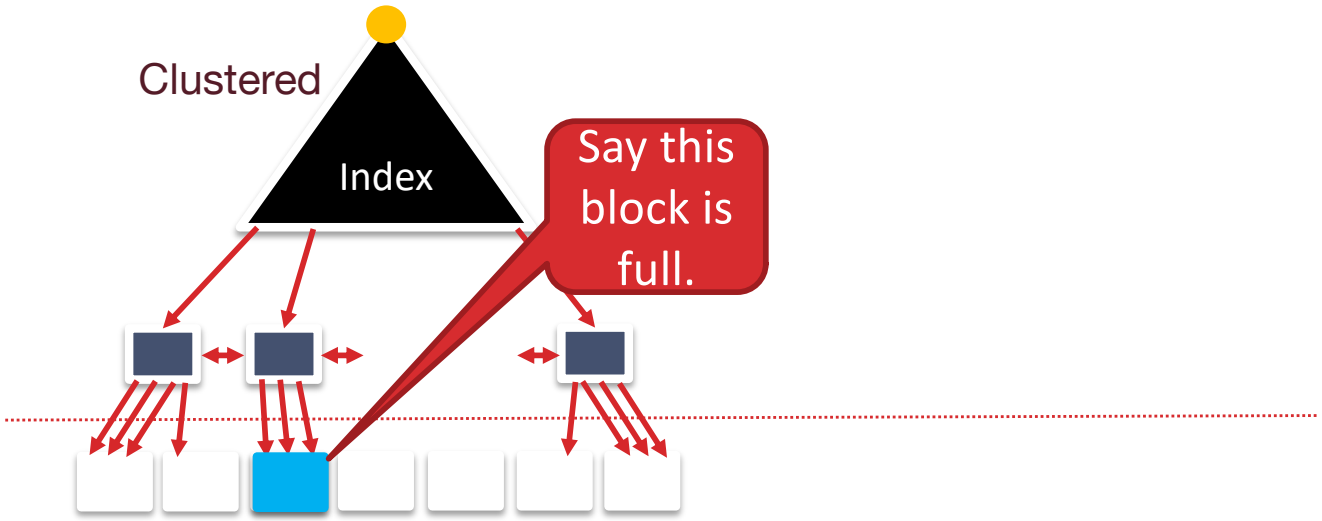


3 Heap file pages vs. 5 pages.

In general unclustered can be arbitrarily bad!

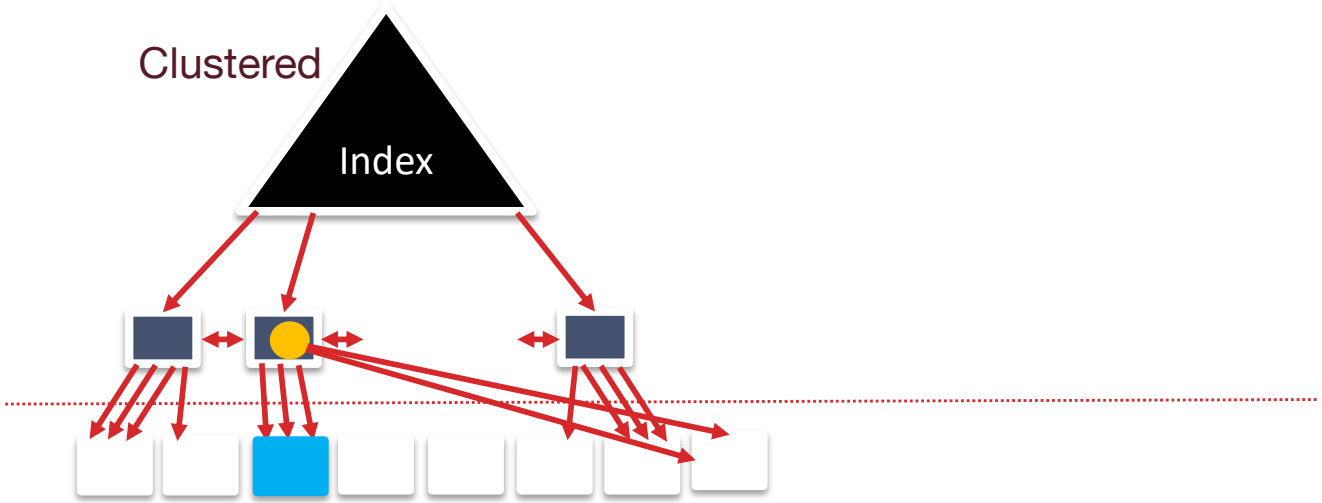
Clustered vs. Unclustered Index Visualization 4

- To build a clustered index, first sort the heap file
  - Leave some free space on each block for future inserts
  - We then try to respect this order "as much as possible"
- Blocks at end of file may be needed for inserts
  - Order of data records is "close to", but not identical to, the sort order



Clustered vs. Unclustered Index Visualization 5

- To build a clustered index, first sort the heap file
  - Leave some free space on each block for future inserts
  - We then try to respect this order "as much as possible"
- Blocks at end of file may be needed for inserts
  - Order of data records is "close to", but not identical to, the sort order



Clustered vs. Unclustered Indexes Pros

- Clustered Index Pros
  - Efficient for range searches due to potential locality benefits
    - Sequential disk access, prefetching, etc.
  - Support certain types of compression
    - More soon on this topic

Clustered vs. Unclustered Indexes Cons

- Clustered Cons
  - More expensive to maintain
    - If we don't maintain, ends up becoming closer to unclustered after many inserts
    - To maintain, we need to periodically update heap file order
      - Can be done on the fly (more expensive per update, but lookup perf is good throughout)
      - Or lazily (less expensive per update but performance can degrade)
  - To reduce cost of maintenance, heap file usually only **packed to 2/3** (or some other fraction <1) to accommodate inserts

B+TREE REFINEMENT: VARIABLE-LENGTH KEYS

Variable Length Keys & Records

- So far we have been using integer keys

- What would happen to our occupancy invariant with variable length keys?

- What about data in leaf pages:

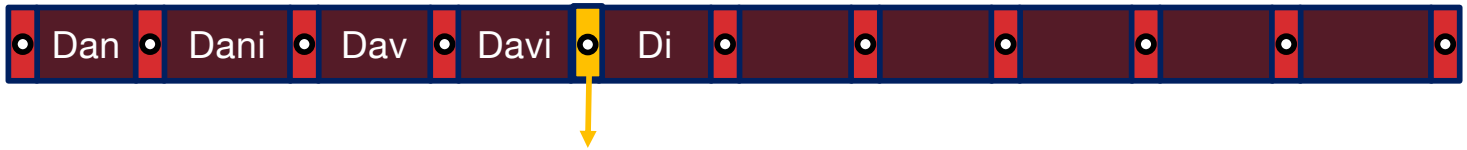
Redefine Occupancy Invariant

- Order (**d**) makes little sense with variable-length entries
  - Different nodes have different numbers of entries.
  - **Index pages** often hold many **more entries** than leaf pages
  - Even with fixed length fields, Alternatives 1 and 3 gives variable length data entries
- Use a physical criterion in practice: ***at-least half-full***
  - Measured in **bytes**
- Many real systems are even sloppier than this
  - Only reclaim space when a page is completely empty.
  - Basically the deletion policy we described earlier…

Prefix Compress Keys?

- How can we get more keys on a page?

- What if we compress the keys?

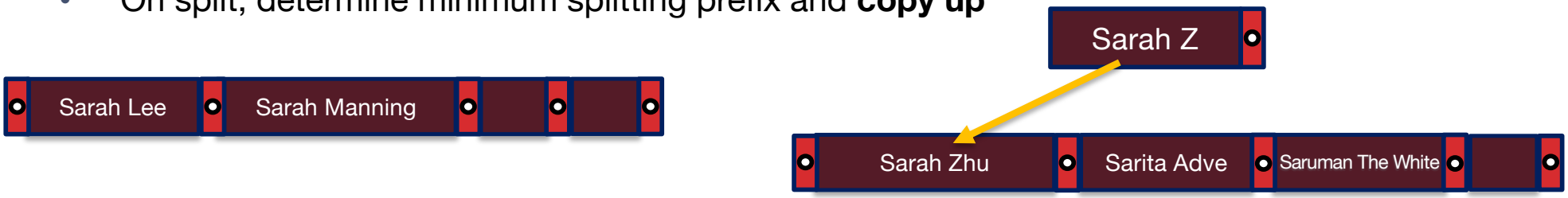


- Are these the same?
  - David Jones?
  - Not the same partitioning of possible keys
  - But why would we care??

Prefix Key Compression

- What if we compress starting at leaf:

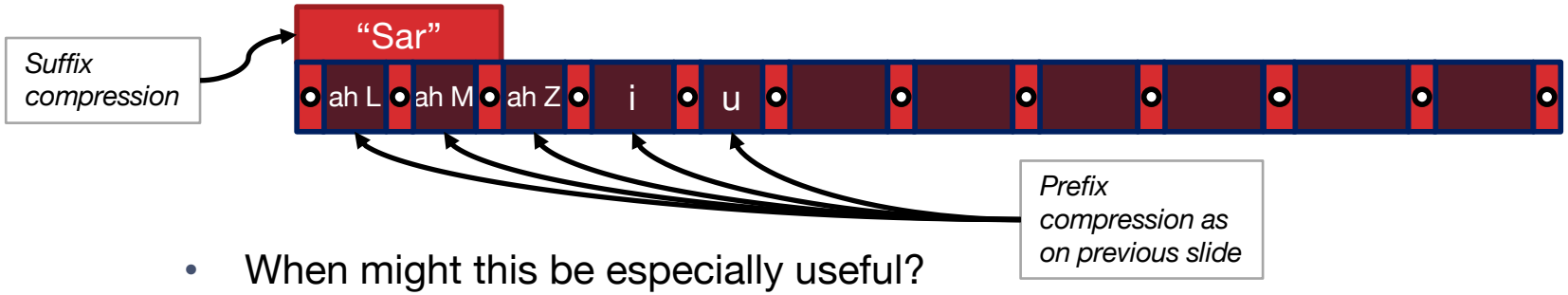
- On split, determine minimum splitting prefix and **copy up**



Suffix Key Compression

- All keys have large common prefix

- Move common prefix to header, leave only (compressed) suffix next to pointer



- When might this be especially useful?
  - Composite Keys. Example?
    - <Zip code, Last Name, First Name>

B+-TREE COSTS

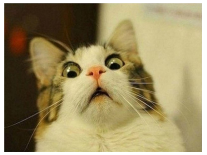
Recall: Cost of Operations

- **Can we do better with indexes?**

- **B:** Number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

Recall we are interested in the average case

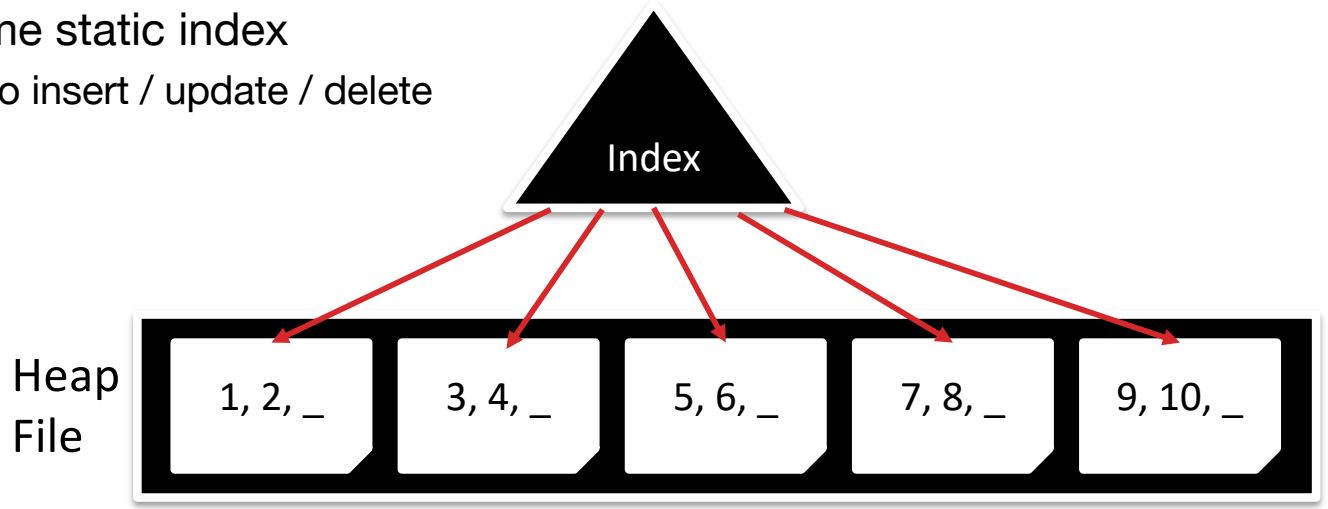
Both reading and writing to the disk cost I/Os!!



	Heap File	Sorted File
Scan all records	B*D	B*D
Equality Search	0.5*B*D	$(\log_2 B)*D$
Range Search	B*D	$((\log_2 B)+pages))*D$
Insert	2*D	$((\log_2 B) + B)*D$
Delete	(0.5*B+1)*D	$((\log_2 B) + B)*D$

Index Assumptions

- Store data by reference (Alternative 2)
- **Clustered** index with 2/3 full heap file pages
  - Clustered → Heap file is initially sorted
  - **Fan-out** (F) (i.e., branching factor) of tree internal node:
    - Page of <key, pointer> pairs ~ O(R) [R: Number of records per block]
    - in practice this is relatively large. Why?
  - Assume static index
    - No insert / update / delete



Cost of Operations

- **Can we do better with indexes?**

- **B:** Number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

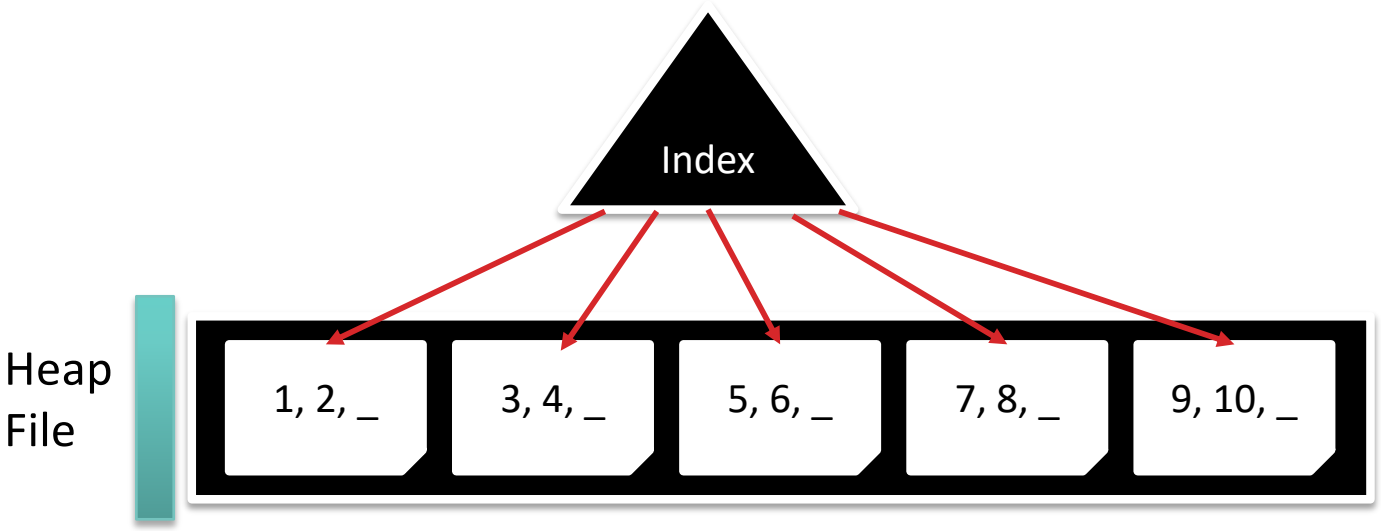
	Heap File	Sorted File	Clustered Index
Scan all records	B*D	B*D	
Equality Search	0.5*B*D	$(\log_2 B)*D$	
Range Search	B*D	$((\log_2 B)+pages))*D$	
Insert	2*D	$((\log_2 B) + B)*D$	
Delete	(0.5*B+1)*D	$((\log_2 B) + B)*D$	

Scan all the Records

- Do we need an Index?
  - No
- Cost? = 1.5 * B * D
  - Why?

B: Number of data blocks
D: Average time to read/write disk block

Recall assumption on clustered indexes: heap file pages are only **2/3** full.



Cost of Operations: Scan

	Heap File	Sorted File	Clustered Index
Scan all records	B*D	B*D	3/2 * B * D
Equality Search	0.5*B*D	$(\log_2 B)*D$	
Range Search	B*D	$((\log_2 B)+pages))*D$	
Insert	2*D	$((\log_2 B) + B)*D$	
Delete	(0.5*B+1)*D	$((\log_2 B) + B)*D$	

- **B:** Number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

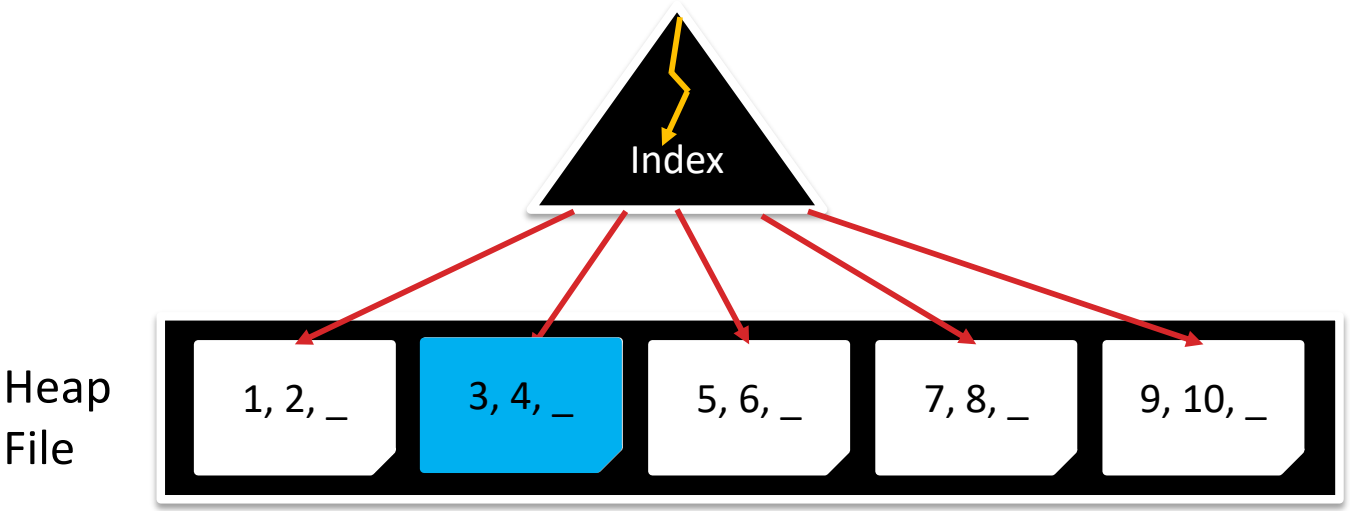
Cost of Operations: Equality Search?

	Heap File	Sorted File	Clustered Index
Scan all records	B*D	B*D	3/2 * B * D
Equality Search	0.5*B*D	$(\log_2 B)*D$	
Range Search	B*D	$((\log_2 B)+pages))*D$	
Insert	2*D	$((\log_2 B) + B)*D$	
Delete	(0.5*B+1)*D	$((\log_2 B) + B)*D$	

- **B:** Number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block
- **F:** Average internal node fanout
- **E:** Average # data entries per leaf

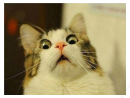
Find the record with key 3

- Two steps
  1. Search index to find the page and record-id
  2. Fetch record-id from heap file



Find the record with key 3

- I/Os for index search = height of index + 1
- = $\log_F$ (# of leaves) + 1 = $\log_F$ (B*R/E) + 1
  - B*R = total number of records, so B*R/E = # of leaves
  - Why +1? Catches the cost of reading the root
    - E.g., F = 4, BR/E = 16
    - Tree height: $\text{Log}_4(16) = 2$
    - But total I/O is 3!
      root → intermediate → leaf

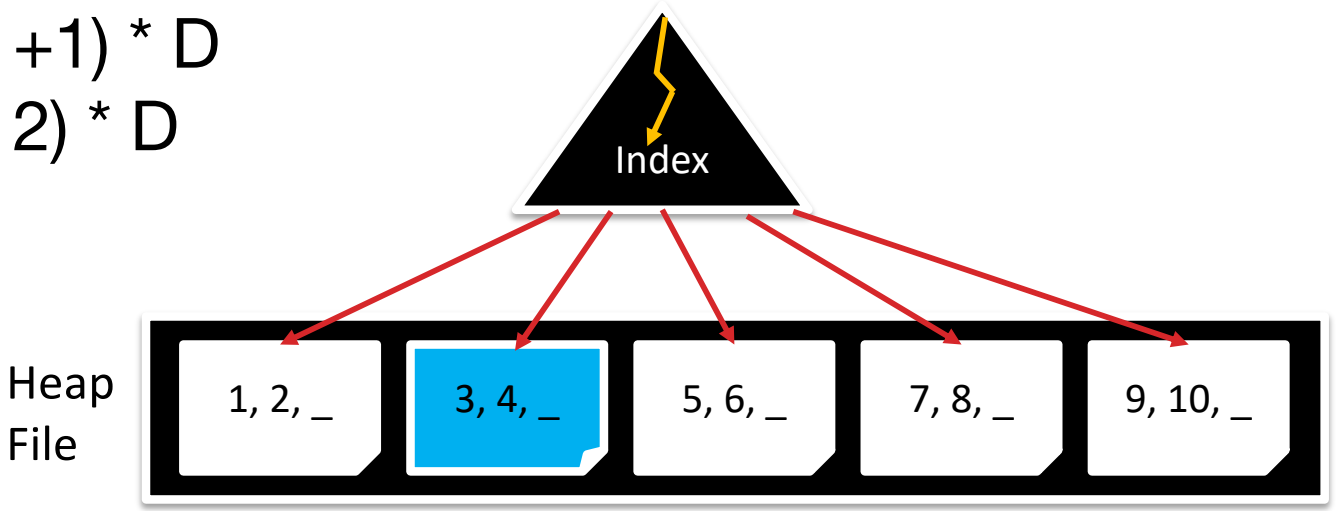


B: # of data blocks
R: # of records per block
D: Avg time to read/write disk block
F: Avg internal node fanout
E: Avg # data entries per leaf

Find the record with key 3

- I/Os for lookup record in heap file by record-id: 1
  - Recall record-id = <page, slot #>

- So total cost: (# of I/Os) * D
  = (I/Os for index search + I/Os for heap file read) * D
  = ($\log_F$ (BR/E)+1+1) * D
  = ($\log_F$ (BR/E) + 2) * D



Cost of Operations: Equality Search

	Heap File	Sorted File	Clustered Index
Scan all records	B*D	B*D	3/2 * B * D
Equality Search	0.5*B*D	$(\log_2 B)*D$	$(\log_F(BR/E)+2)*D$
Range Search	B*D	$((\log_2 B)+pages))*D$	
Insert	2*D	$((\log_2 B) + B)*D$	
Delete	(0.5*B+1)*D	$((\log_2 B) + B)*D$	

- **B:** Number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block
- **F:** Average internal node fanout
- **E:** Average # data entries per leaf

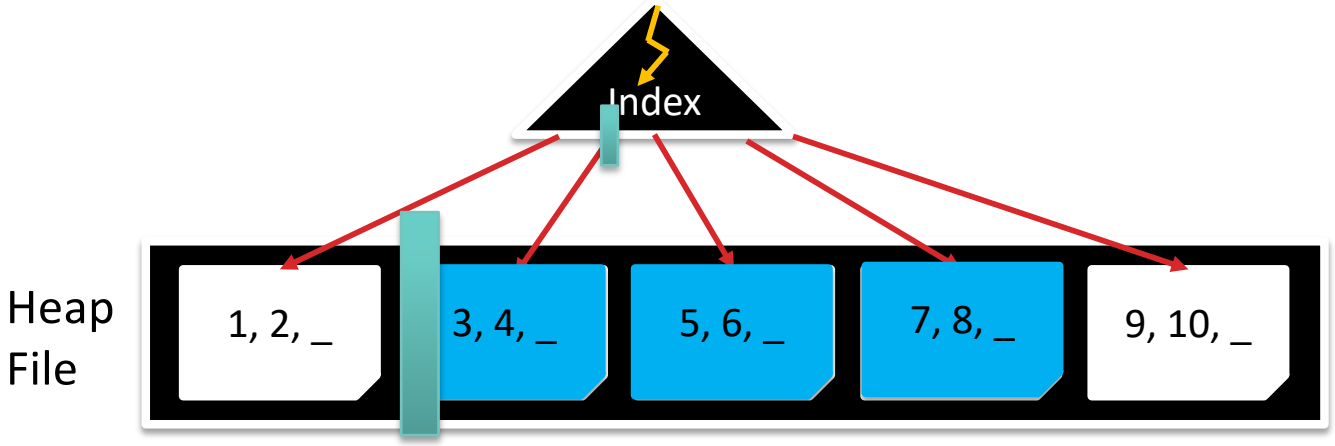
Cost of Operations: Range Search?

	Heap File	Sorted File	Clustered Index
Scan all records	B*D	B*D	3/2 * B * D
Equality Search	0.5*B*D	$(\log_2 B)*D$	$(\log_F(BR/E)+2)*D$
Range Search	B*D	$((\log_2 B)+pages))*D$	
Insert	2*D	$((\log_2 B) + B)*D$	
Delete	(0.5*B+1)*D	$((\log_2 B) + B)*D$	

- **B:** Number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block
- **F:** Average internal node fanout
- **E:** Average # data entries per leaf

Find records with keys between 3 and 7

- Three steps:
  1. Search index to find first page to read
  2. Scan index leaf pages to find which heap file pages to read
  3. Read the corresponding records from heap file
- I/Os for 1: $\log_F (B*R/E) + 1$ [+1 for the index leaf page]
- I/Os for 3: (3/2 * #pages) [#pages: # of pages storing records between 3 and 7]
- I/Os for 2: (3/2 * #pages) [over-approximate and assume same as 2]
- Total cost : $((\log_F (B*R/E) + 1) + 2 * (3/2 * \#pages) - 1) * D = (\log_F (B*R/E) + 3 * \#pages) * D$
  - Why -1? We overcounted accessing the first leaf page in the index



Cost of Operations: Range Search

	Heap File	Sorted File	Clustered Index
Scan all records	B*D	B*D	3/2 * B * D
Equality Search	0.5*B*D	$(\log_2 B)*D$	$(\log_F(BR/E)+2)*D$
Range Search	B*D	$((\log_2 B)+pages))*D$	$(\log_F(BR/E)+3*pages)*D$
Insert	2*D	$((\log_2 B) + B)*D$	
Delete	(0.5*B+1)*D	$((\log_2 B) + B)*D$	

- **B:** Number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block
- **F:** Average internal node fanout
- **E:** Average # data entries per leaf

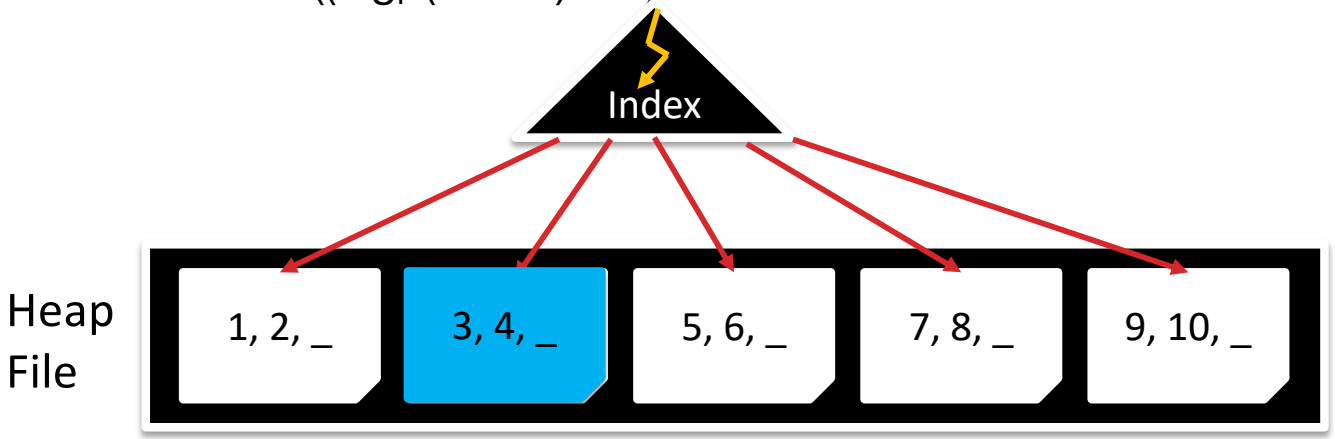
Cost of Operations: Insert?

	Heap File	Sorted File	Clustered Index
Scan all records	B*D	B*D	3/2 * B * D
Equality Search	0.5*B*D	$(\log_2 B)*D$	$(\log_F(BR/E)+2)*D$
Range Search	B*D	$((\log_2 B)+pages))*D$	$(\log_F(BR/E)+3*pages)*D$
Insert	2*D	$((\log_2 B) + B)*D$	
Delete	(0.5*B+1)*D	$((\log_2 B) + B)*D$	

- **B:** Number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block
- **F:** Average internal node fanout
- **E:** Average # data entries per leaf

Insert record with key 4.5

- Three steps:
  1. Search index to find heap file page to modify
  2. Read corresponding page and modify in memory
  3. Write back the modified index leaf and heap file pages
- I/Os for 1: $\log_F (B*R/E) + 1$
- I/Os for 2: 1
- I/Os for 3: 2 [1 for index leaf, 1 for heap file page]
- Total cost : $((\log_F (B*R/E) + 4) * D$



Cost of Operations: Insert

	Heap File	Sorted File	Clustered Index
Scan all records	B*D	B*D	3/2 * B * D
Equality Search	0.5*B*D	$(\log_2 B)*D$	$(\log_F(BR/E)+2)*D$
Range Search	B*D	$((\log_2 B)+pages))*D$	$(\log_F(BR/E)+3*pages)*D$
Insert	2*D	$((\log_2 B) + B)*D$	$(\log_F(BR/E)+4)*D$
Delete	(0.5*B+1)*D	$((\log_2 B) + B)*D$	

- **B:** Number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block
- **F:** Average internal node fanout
- **E:** Average # data entries per leaf

Cost of Operations: Delete

	Heap File	Sorted File	Clustered Index
Scan all records	B*D	B*D	3/2 * B * D
Equality Search	0.5*B*D	$(\log_2 B)*D$	$(\log_F(BR/E)+2)*D$
Range Search	B*D	$((\log_2 B)+pages))*D$	$(\log_F(BR/E)+3*pages)*D$
Insert	2*D	$((\log_2 B) + B)*D$	$(\log_F(BR/E)+4)*D$
Delete	(0.5*B+1)*D	$((\log_2 B) + B)*D$	$(\log_F(BR/E)+4)*D$

- **B:** Number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block
- **F:** Average internal node fanout
- **E:** Average # data entries per leaf

Cost of Operations: Big O Notation

	Heap File	Sorted File	Clustered Index
Scan all records	O(B)	O(B)	O(B)
Equality Search	O(B)	$O(\log_2 B)$	$O(\log_F B)$
Range Search	O(B)	$O(\log_2 B)$	$O(\log_F B)$
Insert	O(1)	O(B)	$O(\log_F B)$
Delete	O(B)	O(B)	$O(\log_F B)$

- **B:** Number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block
- **F:** Average internal node fanout
- **E:** Average # data entries per leaf

Constant factors matter!

- Assume you can do 100 sequential I/Os in the time of 1 random I/O

- For a particular lookup, is a B+-tree better than a full-table scan?
  - Better be very “selective”!
    - Visit < ~1% of pages!
  - Two ways to make that happen:
    - Use a clustered index so that most reads are sequential
    - Use SSD so that random and sequential reads have the same cost

Summary

- Query Structure
  - Understand composite search keys
  - Lexicographic order and search key prefixes

- Data Storage
  - Data Entries: Alt 1 (tuples), Alt 2 (recordIds), Alt 3 (lists of recordIds)
  - Clustered vs. Unclustered
    - Only Alt 2 & 3!

Summary

- Variable length key refinements
  - Fill factors for variable-length keys
  - Prefix and suffix key compression

- B+-tree Cost Model
  - Attractive big-O
  - But don’t forget constant factors of random I/O
    - Hard to beat sequential I/O of scans unless very selective
  - Indexes beyond B+-trees for more complex searches