Index Files and B+Tree Refinements

R & G - Chapter 9-10

Berkeley cs186

General characteristics of an index: An Outline

- Issues to consider in any index structure (not just B+-trees)
  - Query support: what class of queries does the index allow?
  - Choice of Search Key
    - Affects the queries for which we can use an index.
  - Data Entry Storage
    - Affects performance of the index
  - Variable-length key tricks
    - Affects performance of the index
  - Cost Model for Index vs Heap vs Sorted File

QUERY SUPPORT

Indexes: Basic Selection

- **Basic Selection:** <key> <op> <constant>
  - Equality selections (op is =)
  - Range selections (op is one of <, >, <=, >=, BETWEEN)
  - B+-trees provide both
  - Linear Hash indexes provide only equality (but are interesting!)

Indexes: Other Selections

- **More Exotic Selections:**
  - 2-d box (current map boundaries)
  - 2-d circle (“within 2 miles of Empire State Building”)
  - Common **n-dimensional indexes**: R-tree, KD-tree, etc.
    - Beware of the curse of dimensionality
  - Near-neighbor queries (“10 restaurants closest to Empire State Building”)
  - Regular expression matches, genome string matches, etc.
  - See Postgres’ GiST indexes for a flexible structure developed at Berkeley

The image part with relationship ID rId5 was not found in the file.

Image Url

For Today

- In the remainder of our discussion, we'll focus on traditional 1-d range search
  - And equality as a special case
  - As in B+-trees

Search Key and Ordering

- Can index on any ordered subset of columns. Order matters!
  - Determines the queries supported
- In an ordered index (e.g. B+-tree) the keys are ordered **lexicographically** by the search key columns:
  - Ordered by the 1^{st} column
  - 2 items match on 1^{st} column? Ordered by 2^{nd}
  - Match on 1^{st} and 2^{nd} column? Ordered by 3^{rd}
  - Etc.
- E.g. table to right ordered lexicographically by the search key <Age, Salary>

SSN	Last Name	First Name	Age	Salary
123	Adams	Elmo	31	$300
443	Grouch	Oscar	32	$400
244	Oz	Bert	55	$140
134	Sanders	Ernie	55	$400

Search Key and Ordering, Pt 2.

- Defn: A **composite search key** on columns (k_1, k_2, …, k_n) "matches" a query if:
  - The query is a *conjunction* of *m* >= *0* equality clauses of the form:
    $k_1 = <val_1>$ AND $k_2 = <val_2>$ AND .. AND $k_m = <val_m>$
    and at most 1 additional *range* clause of the form:
    AND k_{m+1} op $<val>$, where op is one of $\{<, >\}$

- Why does this "match"? *Lookup and scan in lexicographic order*
  - Can do a lookup on equality conjuncts to find start-of-range
  - Can do a scan of contiguous data entries at leaves
    - satisfy the m+1st conjunct
    - or if there is no m+1st conjunct
      - scan the entire set of matches to the first m conjuncts

Search Key and Ordering, Pt 3

- **Composite Keys:** more than one column
  - **Lexicographic order**
  - Search a *range?*
  - <Age, Salary>

SSN	Last Name	First Name	Age	Salary
123	Adams	Elmo	31	$300
443	Grouch	Oscar	32	$400
244	Oz	Bert	55	$140
134	Sanders	Ernie	55	$400
176	Grump	Donald	79	$300

- Legend

Green for rows we visit that are in the range

Red for rows we visit that are not in the range

Search Key and Ordering, Pt 4

- **Composite Keys:** more than one column
  - **Lexicographic order**
  - Search a *range?*
  - <Age, Salary>:
    - Age = 31 & Salary = 400

SSN	Last Name	First Name	Age	Salary
123	Adams	Elmo	31	$300
443	Grouch	Oscar	32	$400
244	Oz	Bert	55	$140
134	Sanders	Ernie	55	$400
176	Grump	Donald	79	$300

Search Key and Ordering, Pt 5

- **Composite Keys:** more than one column
  - **Lexicographic order**
  - Search a *range?*
  - <Age, Salary>:
    - ✓ Age = 31 & Salary = 400

SSN	Last Name	First Name	Age	Salary
123	Adams	Elmo	31	$300
443	Grouch	Oscar	32	$400
244	Oz	Bert	55	$140
134	Sanders	Ernie	55	$400
176	Grump	Donald	79	$300

Search Key and Ordering, Pt 6

- **Composite Keys:** more than one column
  - **Lexicographic order**
  - Search a *range?*
  - <Age, Salary>:
    - ✓ Age = 31 & Salary = 400
    - Age = 55 & Salary > 200

SSN	Last Name	First Name	Age	Salary
123	Adams	Elmo	31	$300
443	Grouch	Oscar	32	$400
244	Oz	Bert	55	$140
134	Sanders	Ernie	55	$400
176	Grump	Donald	79	$300

Search Key and Ordering, Pt 6, cont

- **Composite Keys:** more than one column
  - **Lexicographic order**
  - Search a *range?*
  - <Age, Salary>:
    - Age = 31 & Salary = 400
    - Age = 55 & Salary > 200

SSN	Last Name	First Name	Age	Salary
123	Adams	Elmo	31	$300
443	Grouch	Oscar	32	$400
244	Oz	Bert	55	$140
134	Sanders	Ernie	55	$400
176	Grump	Donald	79	$300

Search Key and Ordering, Pt. 7

- **Composite Keys:** more than one column
  - **Lexicographic order**
  - Search a *range?*
  - <Age, Salary>:
    - ✓ Age = 31 & Salary = 400
    - ✓ Age = 55 & Salary > 200
    - Age > 31 & Salary = 400

SSN	Last Name	First Name	Age	Salary
123	Adams	Elmo	31	$300
443	Grouch	Oscar	32	$400
244	Oz	Bert	55	$140
134	Sanders	Ernie	55	$400
176	Grump	Donald	79	$300

Search Key and Ordering, Pt 8

- **Composite Keys:** more than one column
  - **Lexicographic order**
  - Search a *range?*
  - <Age, Salary>:
    - ✓ Age = 31 & Salary = 400
    - ✓ Age = 55 & Salary > 200
    - ✗ Age > 31 & Salary = 400

SSN	Last Name	First Name	Age	Salary
123	Adams	Elmo	31	$300
443	Grouch	Oscar	32	$400
244	Oz	Bert	55	$140
134	Sanders	Ernie	55	$400
176	Grump	Donald	79	$300

✗ Not a lexicographic range. Either visits useless rows or has to “bounce through” the index.

Search Key and Ordering, Pt 9

- **Composite Keys:** more than one column
  - **Lexicographic order**
  - Search a *range?*
  - <Age, Salary>:
    - ✓ Age = 31 & Salary = 400
    - ✓ Age = 55 & Salary > 200
    - ✗ Age > 31 & Salary = 400
    - Age = 31

SSN	Last Name	First Name	Age	Salary
123	Adams	Elmo	31	$300
443	Grouch	Oscar	32	$400
244	Oz	Bert	55	$140
134	Sanders	Ernie	55	$400
176	Grump	Donald	79	$300

✗ Not a lexicographic range. Either visits useless rows or has to “bounce through” the index.

Search Key and Ordering, Pt 10

- **Composite Keys:** more than one column
  - **Lexicographic order**
  - Search a *range?*
  - <Age, Salary>:
    - Age = 31 & Salary = 400
    - Age = 55 & Salary > 200
    - Age > 31 & Salary = 400
    - Age = 31

SSN	Last Name	First Name	Age	Salary
123	Adams	Elmo	31	$300
443	Grouch	Oscar	32	$400
244	Oz	Bert	55	$140
134	Sanders	Ernie	55	$400
176	Grump	Donald	79	$300

Not a lexicographic range. Either visits useless rows or has to "bounce through" the index.

Search Key and Ordering, Pt 11

- **Composite Keys:** more than one column
  - **Lexicographic order**
  - Search a *range?*
  - <Age, Salary>:
    - ✓ Age = 31 & Salary = 400
    - ✓ Age = 55 & Salary > 200
    - ✗ Age > 31 & Salary = 400
    - ✓ Age = 31
    - Age > 31

SSN	Last Name	First Name	Age	Salary
123	Adams	Elmo	31	$300
443	Grouch	Oscar	32	$400
244	Oz	Bert	55	$140
134	Sanders	Ernie	55	$400
176	Grump	Donald	79	$300

✗ Not a lexicographic range. Either visits useless rows or has to "bounce through" the index.

Search Key and Ordering, Pt 12

- **Composite Keys:** more than one column
  - **Lexicographic order**
  - Search a *range?*
  - <Age, Salary>:
    - ✓ Age = 31 & Salary = 400
    - ✓ Age = 55 & Salary > 200
    - ✗ Age > 31 & Salary = 400
    - ✓ Age = 31
    - ✓ Age > 31

SSN	Last Name	First Name	Age	Salary
123	Adams	Elmo	31	$300
443	Grouch	Oscar	32	$400
244	Oz	Bert	55	$140
134	Sanders	Ernie	55	$400
176	Grump	Donald	79	$300

✗ Not a lexicographic range. Either visits useless rows or has to “bounce through” the index.

Search Key and Ordering, Pt 13

- **Composite Keys:** more than one column
  - **Lexicographic order**
  - Search a *range?*
  - <Age, Salary>:
    - ✓ Age = 31 & Salary = 400
    - ✓ Age = 55 & Salary > 200
    - ✗ Age > 31 & Salary = 400
    - ✓ Age = 31
    - ✓ Age > 31
    - Salary = 300 ✗ Not a lexicographic range. Either visits useless rows or has to “bounce through” the index.

SSN	Last Name	First Name	Age	Salary
123	Adams	Elmo	31	$300
443	Grouch	Oscar	32	$400
244	Oz	Bert	55	$140
134	Sanders	Ernie	55	$400
176	Grump	Donald	79	$300

Search Key and Ordering, Pt 14

- **Composite Keys:** more than one column
  - **Lexicographic order**
  - Search a *range?*
  - <Age, Salary>:
    - ✓ Age = 31 & Salary = 400
    - ✓ Age = 55 & Salary > 200
    - ✗ Age > 31 & Salary = 400
    - ✓ Age = 31
    - ✓ Age > 31
    - ✗ Salary = 300

✗ Not a lexicographic range. Either visits useless rows or has to “bounce through” the index.

SSN	Last Name	First Name	Age	Salary
123	Adams	Elmo	31	$300
443	Grouch	Oscar	32	$400
244	Oz	Bert	55	$140
134	Sanders	Ernie	55	$400
176	Grump	Donald	79	$300

Data Entry Storage Intro

- What is the representation of data in the index?
  - Actual data or pointer to the data
- How is the data stored in the data file?
  - Clustered or unclustered with respect to the index
- **Big Impact on Performance**
  - We'll learn each of these next

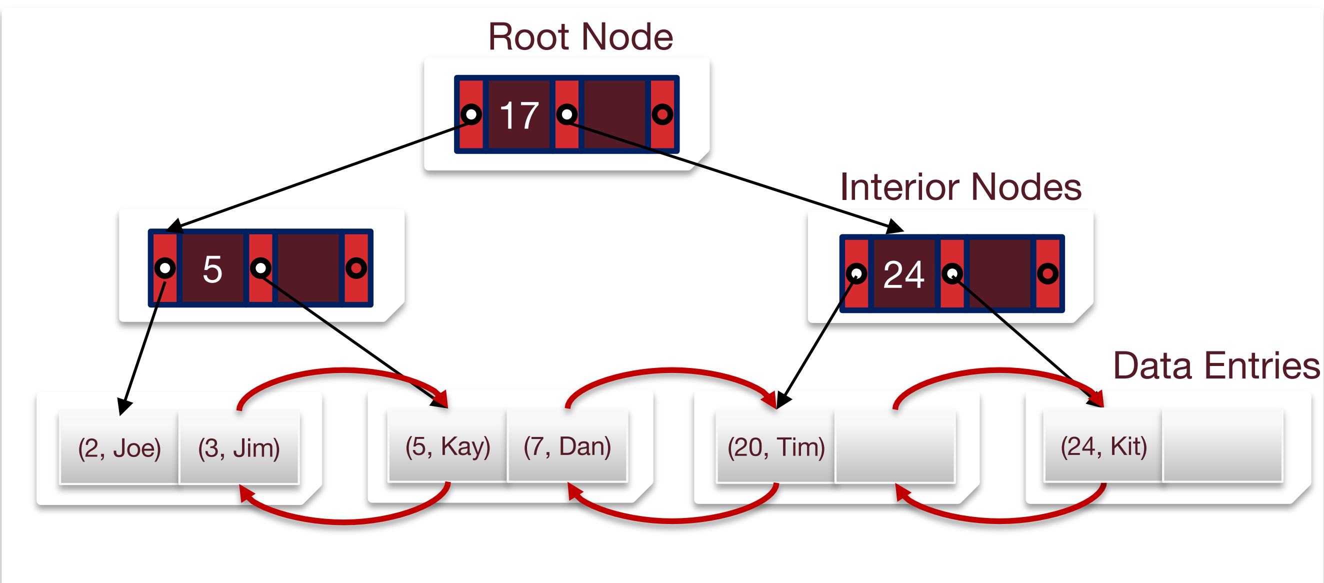
Three basic alternatives for data entries in any index

- Three basic alternatives for data entries in any index
  - Alternative 1: By Value
  - Alternative 2: By Reference
  - Alternative 3: By List of references
    - We’ll look in the context of B+-trees, but applies to any index

Alternative 1 Index (B+ Tree)

- Record contents are stored in the index file
  - No need to follow pointers

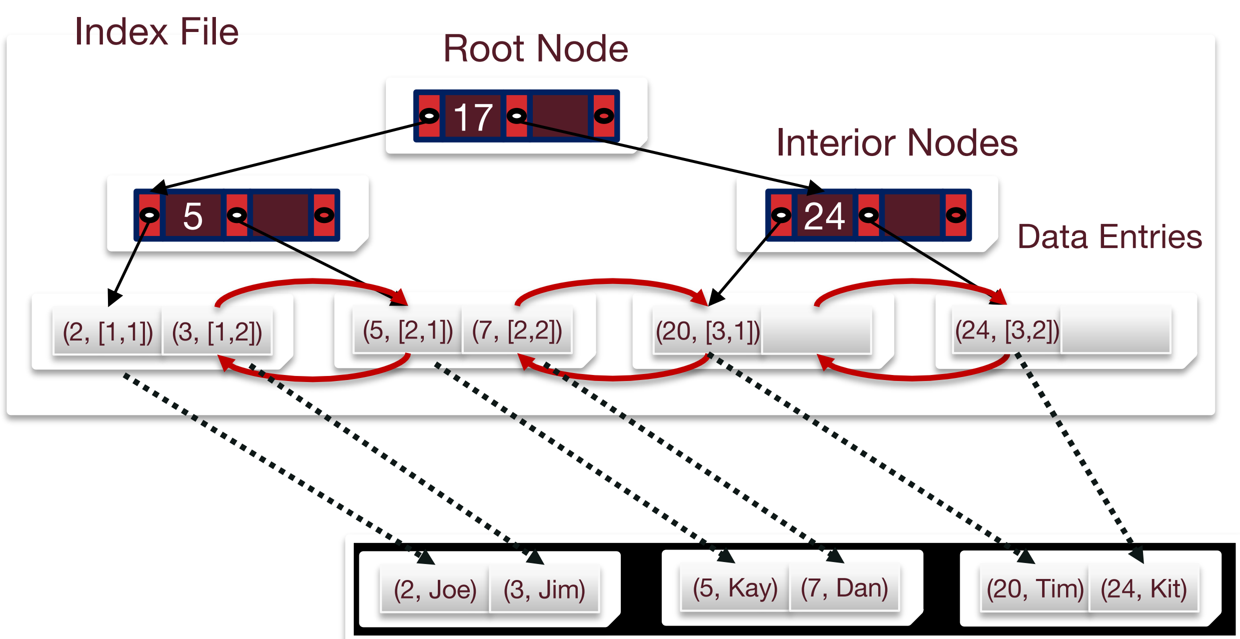
uid	name
2	Joe
3	Jim
5	Kay
7	Dan
20	Tim
24	Kit



Alternative 2 Index

- Alternative 2: **By Reference,** <**k**, rid of matching data record>
  - We used in slides above

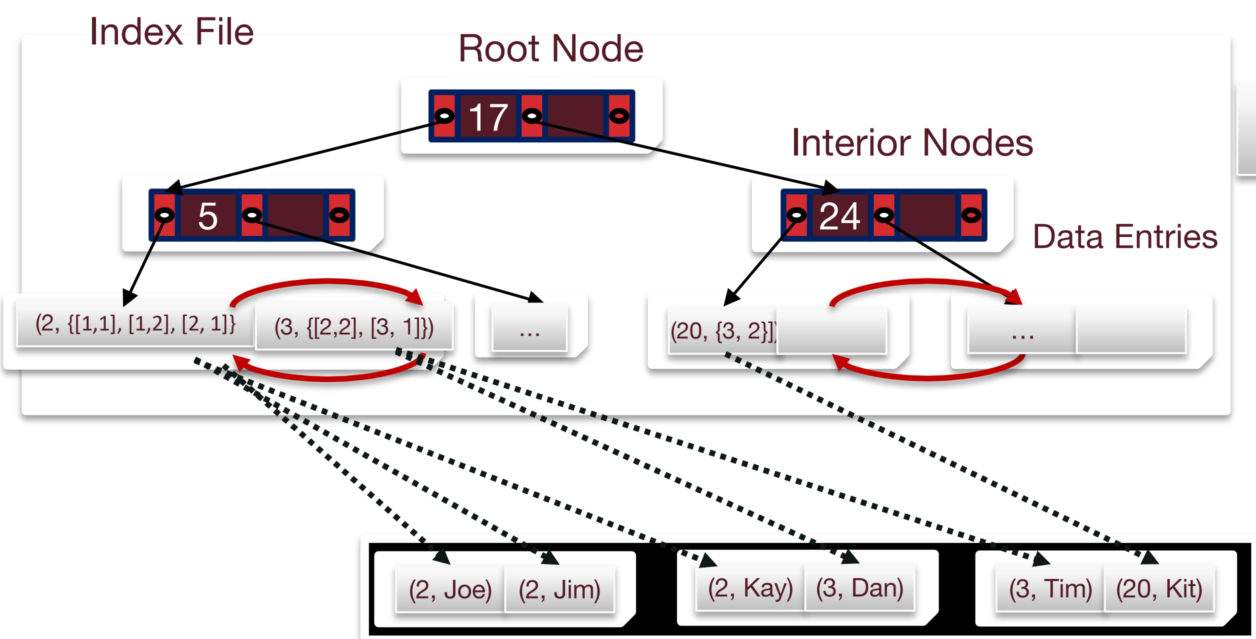
uid	name
2	Joe
3	Jim
5	Kay
7	Dan
20	Tim
24	Kit



Index Contains
(Key, Record Id)
Pairs

Alternative 3 Index

- Alternative 3: **By List of references,** <**k**, list of rids of matching data records>
    - Alternative 3 more compact than alternative 2
        - For very large rid lists, single data entry spans multiple blocks



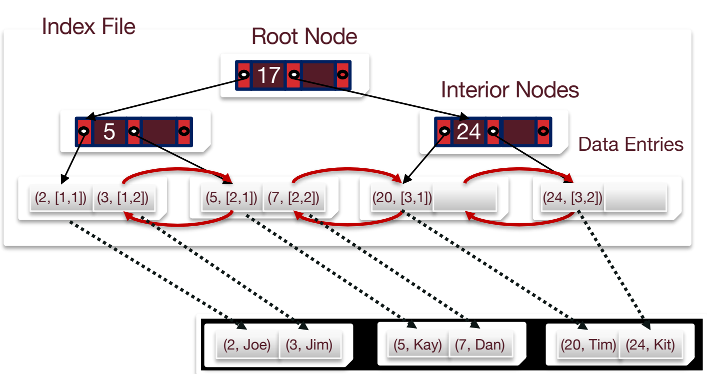
Index Contains
(Key, {list of record Id}) Pairs

Key	Record Id
2	{[1,1], [1,2], [1,3]}
3	4

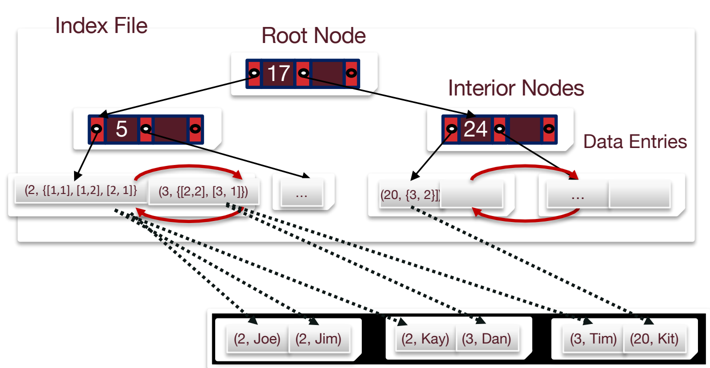
Indexing By Reference

- Both Alternative 2 and Alternative 3 index data *by reference*
- By-reference is *required* to support multiple indexes per table
  - Otherwise we would be replicating entire tuples
  - Replicating data leads to complexity when we're doing updates, so it's something we want to avoid

Alternative 2
Index data entries



Alternative 3
Index data entries



Alternative 2 vs Alternative 3 Table Illustration

Alternative 2
Index data entries

Key	Record Id
Gonzalez	[3, 1]
Gonzalez	[3, 2]
Gonzalez	[3, 3]
Hong	[3, 4]

SSN	Last Name	First Name	Salary
123	Gonzalez	Amanda	$400
443	Gonzalez	Joey	$300
244	Gonzalez	Jose	$140
134	Hong	Sue	$400

Alternative 3
Index data entries

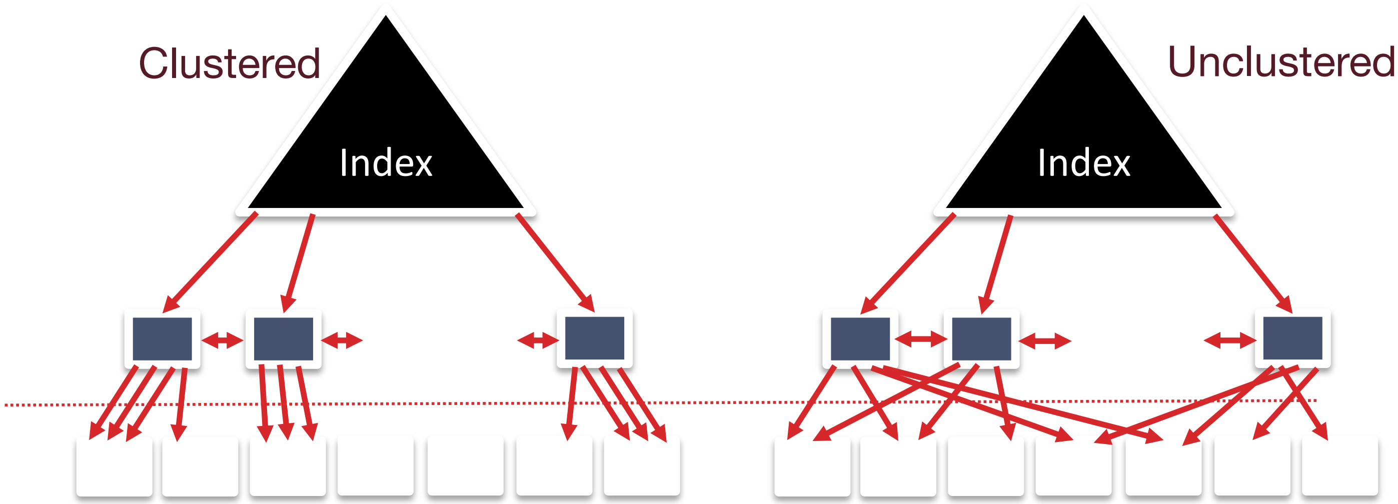
Key	Record Id
Gonzalez	[3, {1, 2, 3}]
Hong	[3,4]

Clustered vs. Unclustered Index

- By-reference indexes (Alt 2 and 3) can be *clustered* or *unclustered*
  - Really this is a property of the heap file associated with the index!
- Clustered index:
  - 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
    - Cost of retrieving data records through index varies greatly based on whether index is clustered or not!
- Note: different definition of “clustering” in AI:
  - grouping nearby items in *n*-space

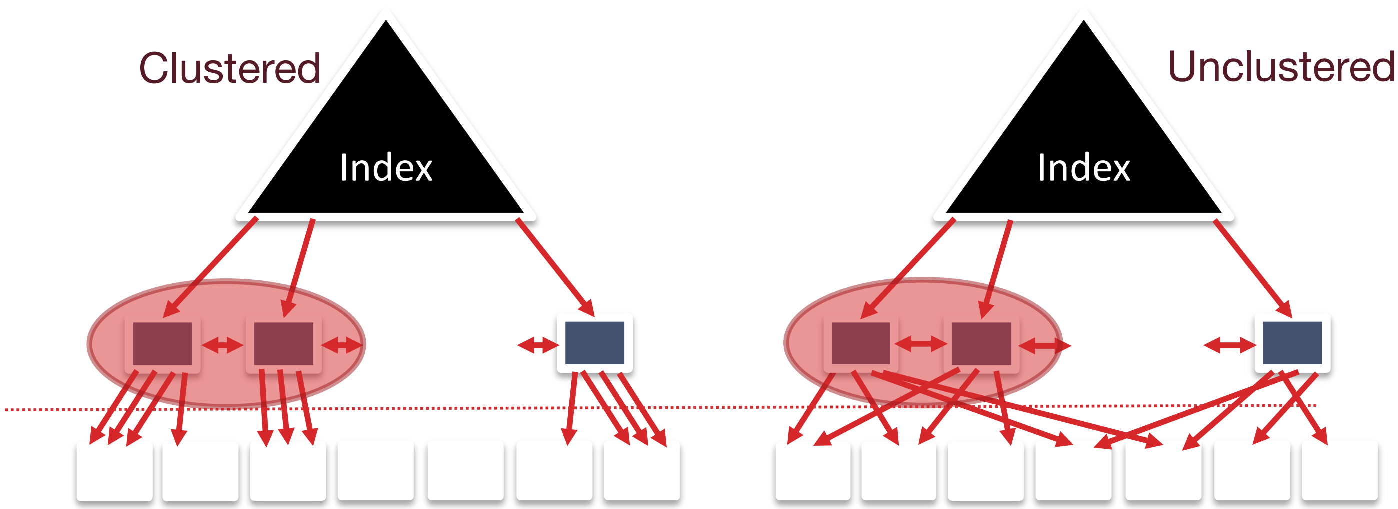
Clustered vs. Unclustered Index Visualization 1

- To build a clustered index, first sort the heap file
  - Leave some free space on each block for future inserts
  - Index entries direct search for data entries



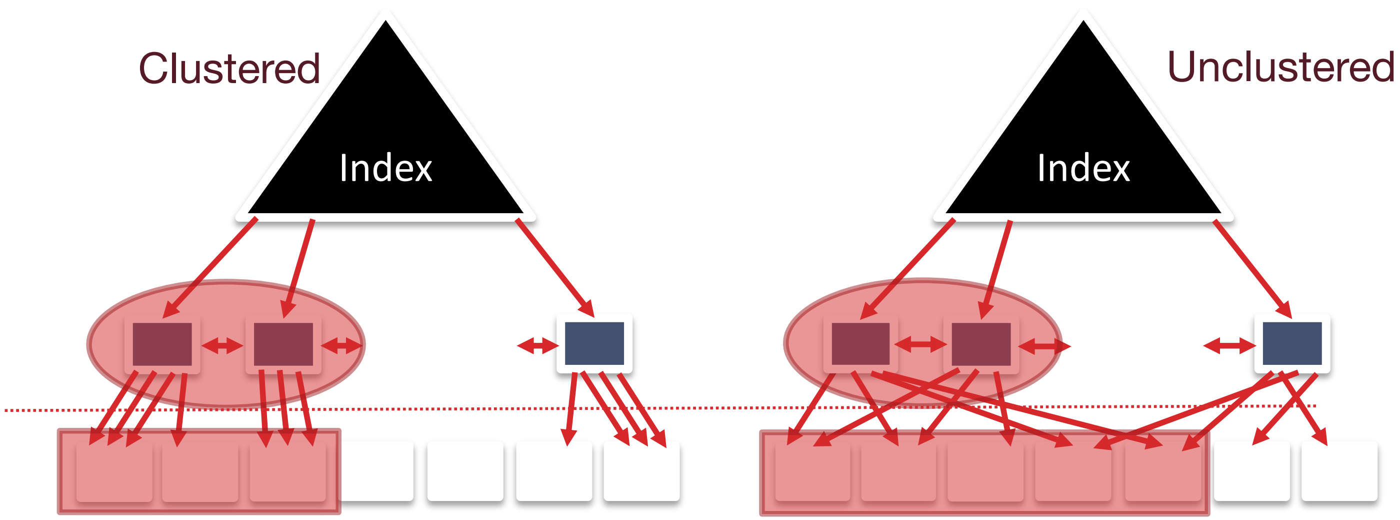
Clustered vs. Unclustered Index Visualization 2

- To build a clustered index, first sort the heap file
  - Leave some free space on each block for future inserts
  - Index entries direct search for data entries



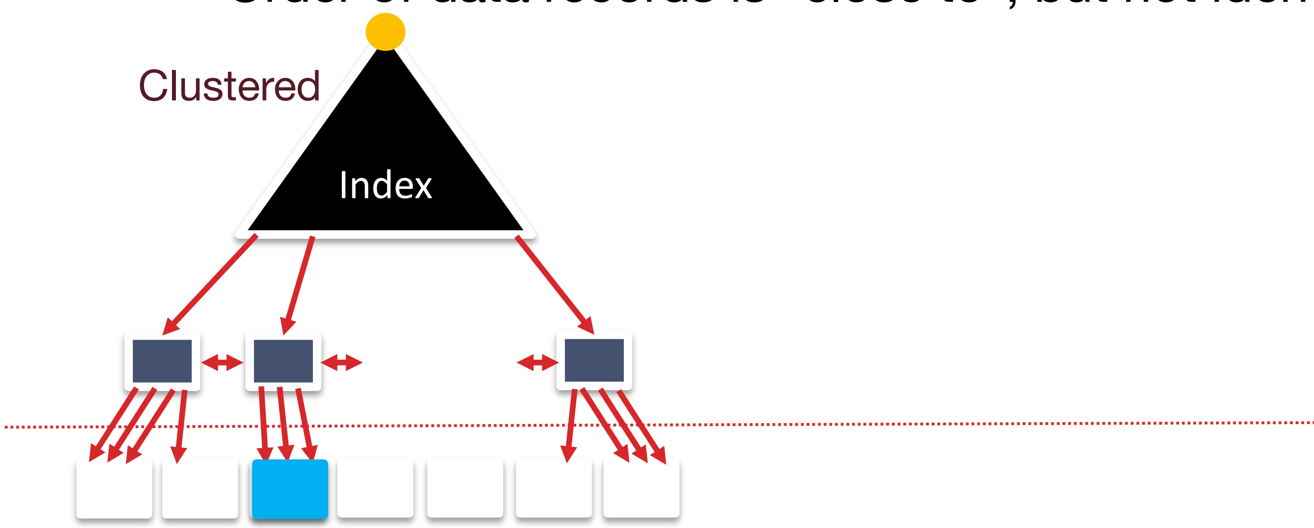
Clustered vs. Unclustered Index Visualization 3

- To build a clustered index, first sort the heap file
  - Leave some free space on each block for future inserts
  - Index entries direct search for data entries



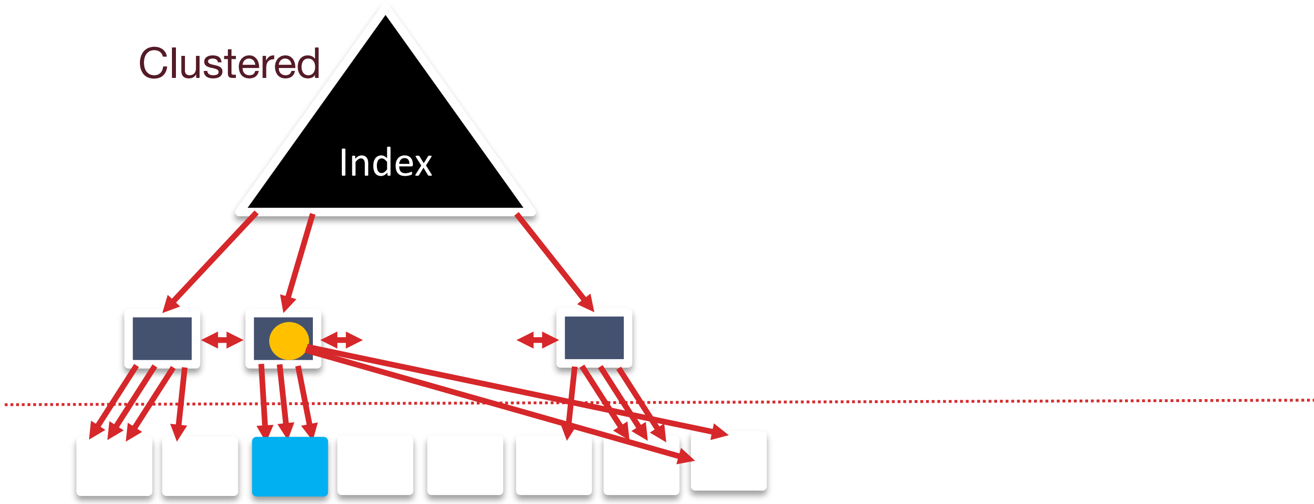
Clustered vs. Unclustered Index Visualization 5

- To build a clustered index, first sort the heap file
    - Leave some free space on each block for future inserts
- 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 6

- To build a clustered index, first sort the heap file
  - Leave some free space on each block for future inserts
- 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
  - 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
    - Need to periodically update heap file order
    - Solution: on the fly or “lazily” via reorganizations
  - Heap file usually only **packed to 2/3** 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:

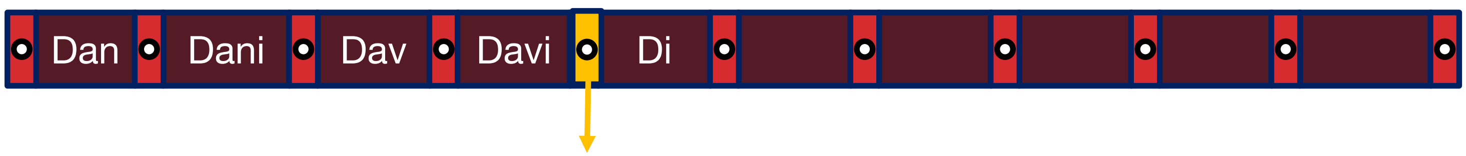
Redefine Occupancy Invariant

- 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, Alternative 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 above…

Prefix Compress Keys?

- How can we get more keys on a page?

- What if we compress the keys?

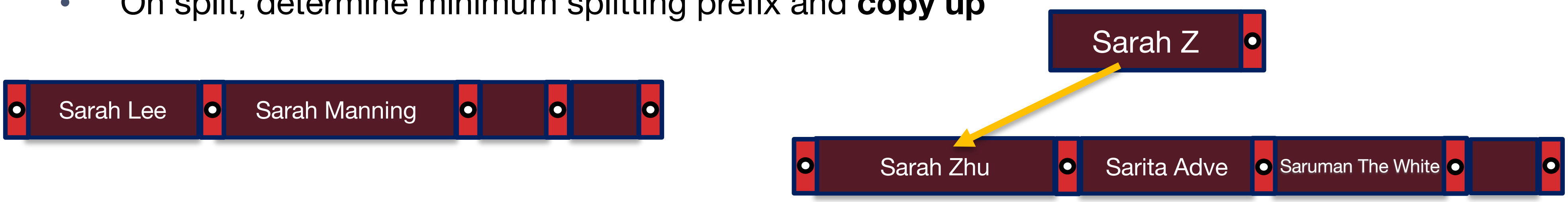


- Are these the same
  - David Jones?
  - Not the same partitioning of possible keys
  - But why would we care??

Prefix Key Compression

- What if we compress starting at leaf:

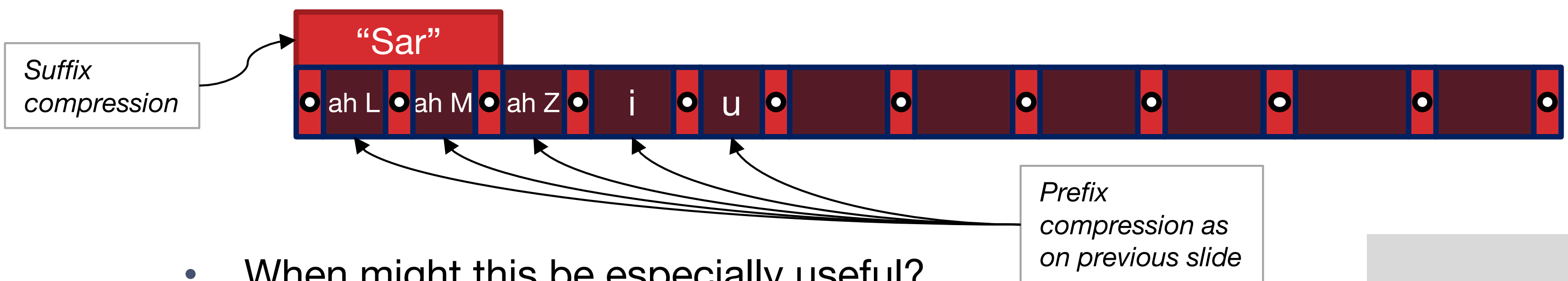
- On split, determine minimum splitting prefix and **copy up**



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

	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

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

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

Cost of Operations

	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	

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

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

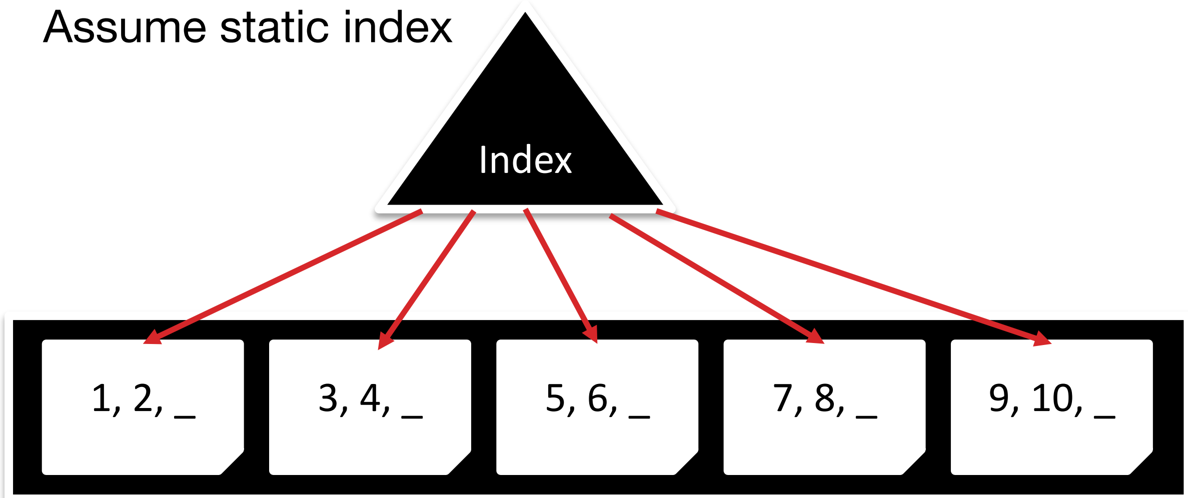
Cost of Operations, cont

	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	

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

Clustered vs. Unclustered Index Assumptions

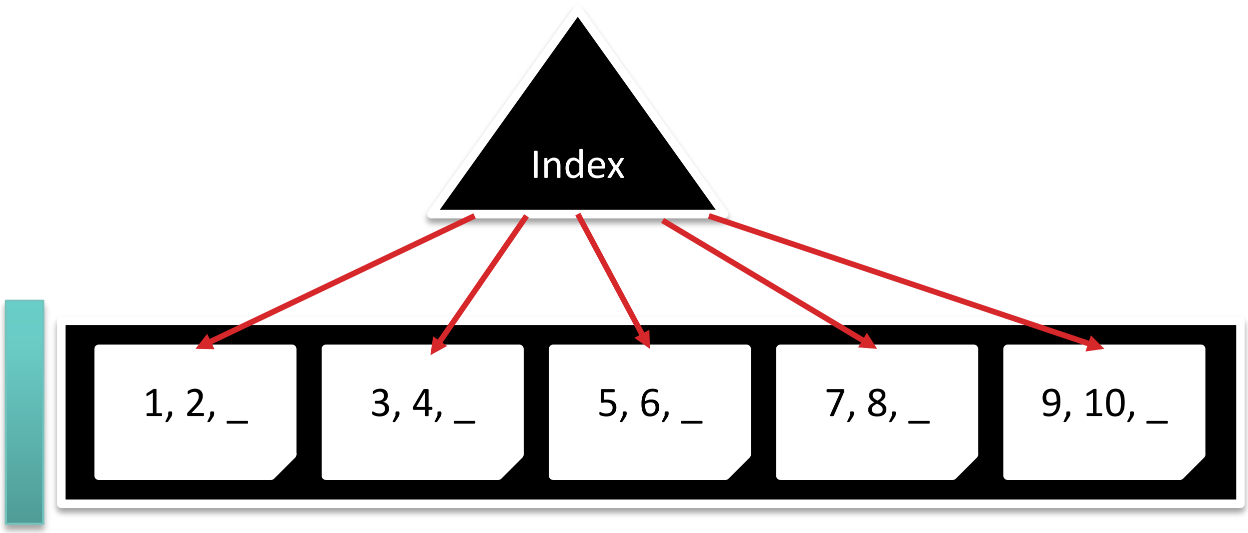
- Store data by reference (Alternative 2)
- Clustered index with 2/3 full heap file pages
    - Clustered → Heap file is initially sorted
    - **Fan-out** (F): relatively large. Why?
        - Page of <key, pointer> pairs ~ O(R)
    - Assume static index



Scan all the Records

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

Recall assumption from before regarding clustered indexes: heap file pages 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:** The number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

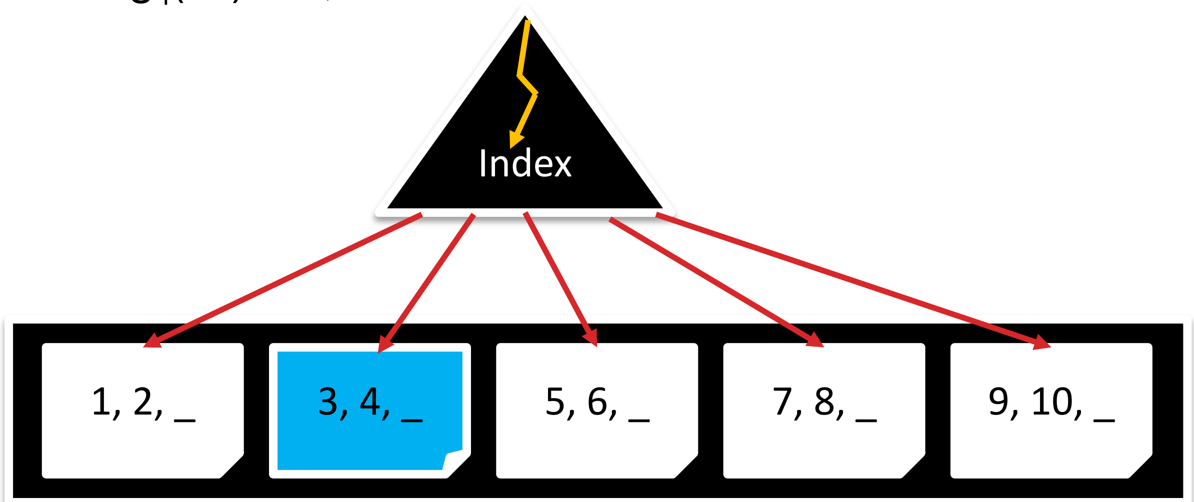
Cost of Operations: Equality Search?

	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:** The 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

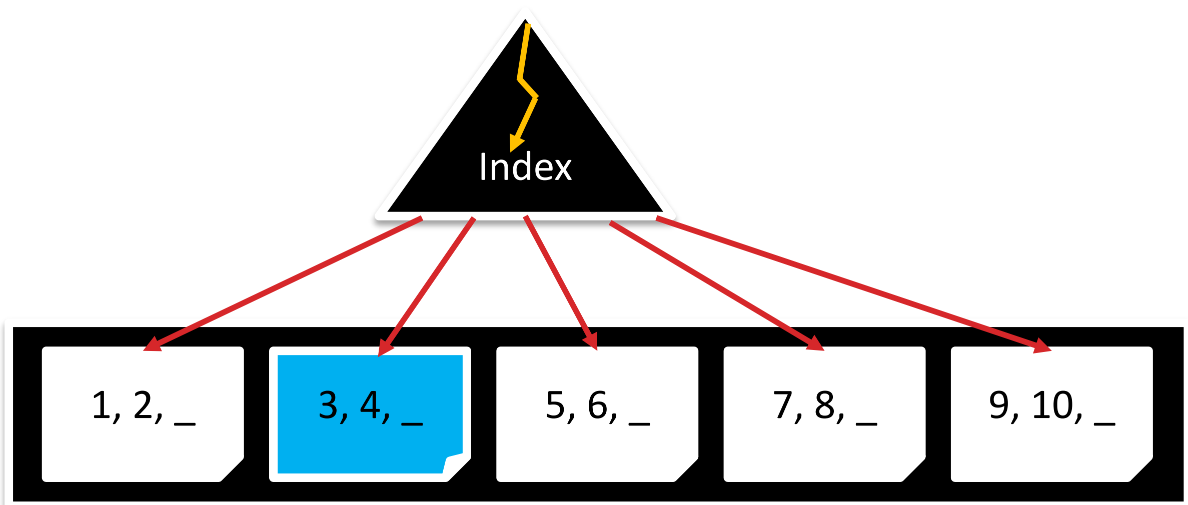
Find the record with key 3, pt 1

- Search the index:= ($\log_F$ (BR/E) + 1) * D
  - BR is the total number of records; E is the #records per leaf
  - the +1 is an "off by 1" thing: catches the cost of the root
  - E.g. F = 4, BR/E = 16: root, intermediate, leaf levels.
  - $\log_4(16) = 2$, and I/O cost is 3!



Find the record with key 3, pt 2

- Search the index:= $(\log_F (BR/E) + 1) * D$
- Lookup record in heap file by record-id = 1 * D
  - Recall record-id = <page, slot #>



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:** The 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

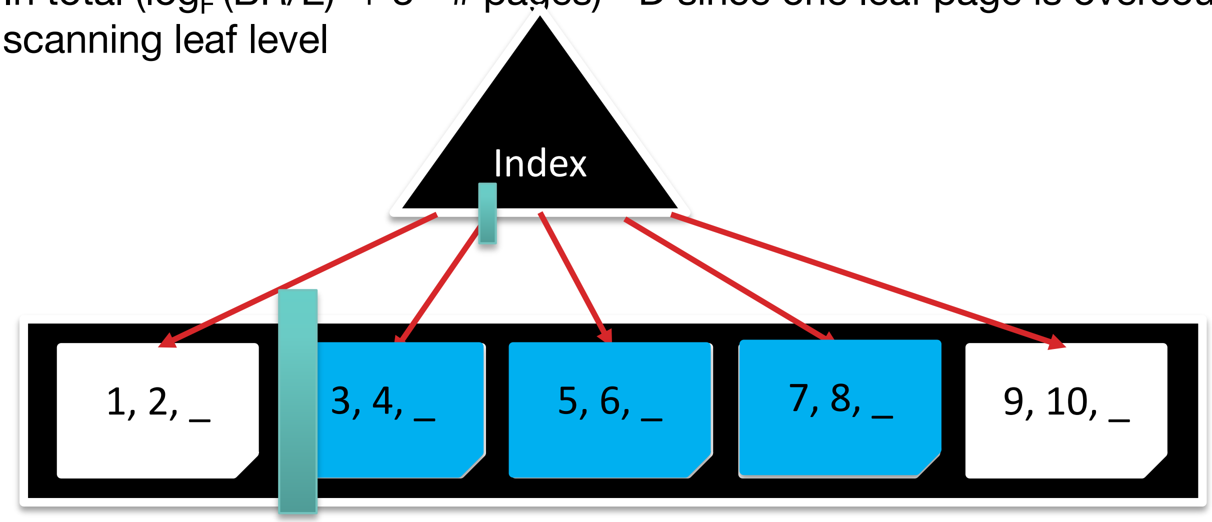
Cost of Operations: Range Search?

	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:** The 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 keys between 3 and 7

- Search the index: = ($\log_F$ (BR/E) + 1) * D
- Scan the leaf level and lookup each matching record in the heap file by record-id
  - Recall record-id = <page, slot #>
- Heap file access: (3/2 * #pages) * D
- Scanning the leaf level is similar to heap file access: assume same (3/2 * #pages) * D
- In total ($\log_F$ (BR/E) + 3 * # pages) * D since one leaf page is overcounted in searching index and scanning leaf level



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:** The 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:** The 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	$(\log_F(BR/E)+4)$*D
Delete	(0.5*B+1)*D	$((\log_2 B) + B)$*D	

- **B:** The 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

Why “+4” in Insert/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:** The 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:** The 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

- 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?
  - Had better be very “selective”
    - Visit < ~1% of pages!
  - Or do mostly sequential I/O at leaf level
    - Clustered index
  - Or use SSD
    - SSDs make indexes attractive
    - Especially for read-mostly workloads

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 Cont

- Variable length key refinements
  - Fill factors for variable-length keys
  - Prefix and suffix key compression
- B+-tree Cost Model
  - Attractive big-O
  - 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