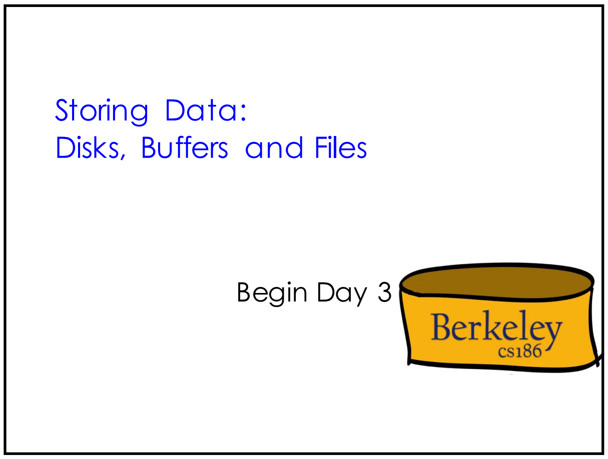
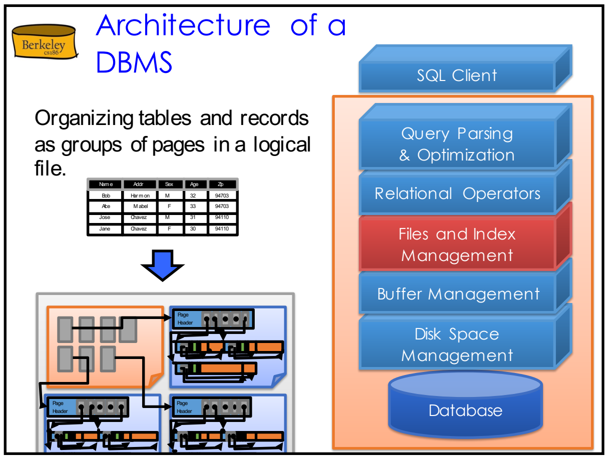


Storing Data: Disks, Buffers and Files

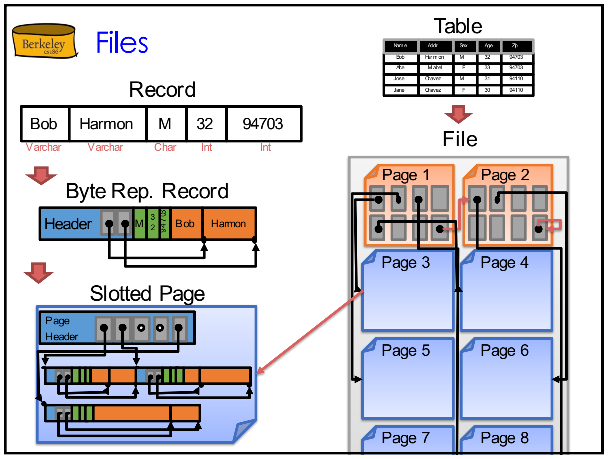
Begin Day 3



Architecture of a DBMS

Organizing tables and records as groups of pages in a logical file.

Name	Addr	Sex	Age	Zip
Bob	Harmon	M	32	94703
Abe	Mabel	F	33	94703
Jose	Chavez	M	31	94110
Jane	Chavez	F	30	94110



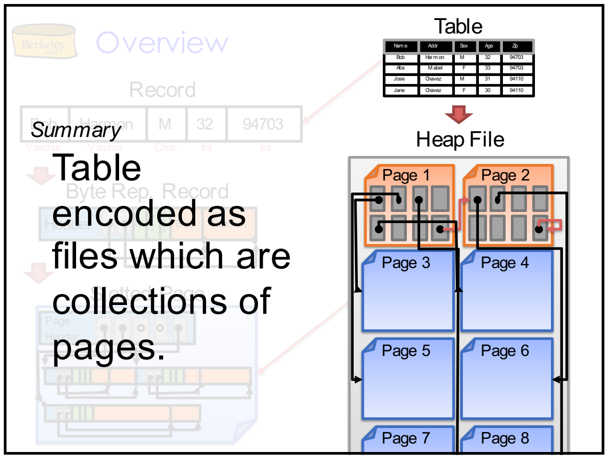
Files

Table

Name	Addr	Sex	Age	Zip
Bob	Harmon	M	32	94703
Abe	Mabel	F	33	94703
Jose	Chavez	M	31	94110
Jane	Chavez	F	30	94110

Record

Bob	Harmon	M	32	94703
Varchar	Varchar	Char	Int	Int

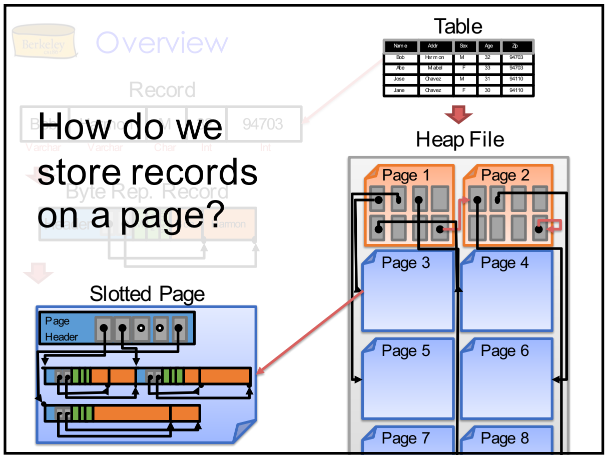


Overview

Summary

Table encoded as files which are collections of pages.

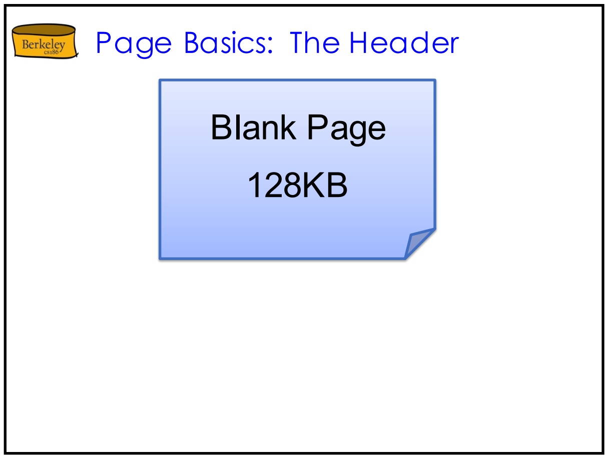
Name	Addr	Sex	Age	Zip
Bob	Harmon	M	32	94703
Abe	Mabel	F	33	94703
Jose	Chavez	M	31	94110
Jane	Chavez	F	30	94110



Overview

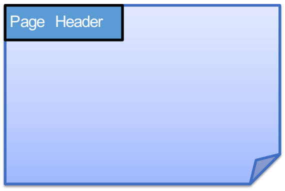
How do we store records on a page?

Name	Addr	Sex	Age	Zip
Bob	Harmon	M	32	94703
Abe	Mabel	F	33	94703
Jose	Chavez	M	31	94110
Jane	Chavez	F	30	94110



Page Basics: The Header

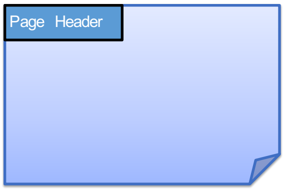
Page Basics: The Header



Header may contain:

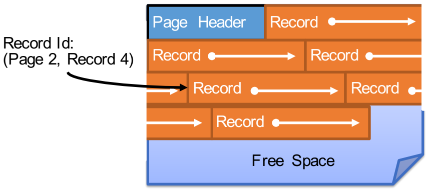
- Number of records
- Free space
- Maybe next/last pointer
- Slot Table … more soon

Things to Address



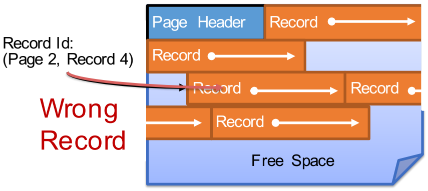
- Record length? *Fixed* or *Variable*
- Find records by record id?
  - record id = (Page, Location in Page)
- How do we add and delete records?

Fixed Length Records: Packed



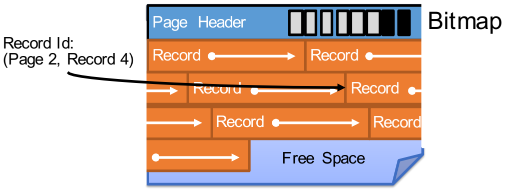
- Pack records densely
- Record id "location in page"?
  - record number in page
- Easy to add: just append
- Delete?

Fixed Length Records: Packed



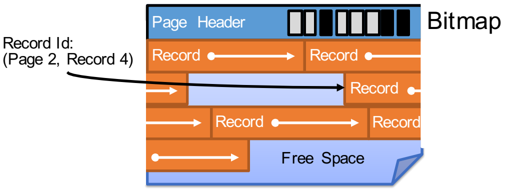
- Pack records densely
- Record id: record number in page
- Easy to add: just append
- Delete? **Packed → re-arrange**←Necessary?

Fixed Length Records: Unpacked



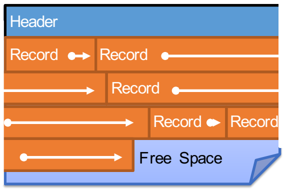
- Bitmap denotes "slots" with records
- Record id: record number in page
- **Insert:** find first empty slot
- **Delete?**

Fixed Length Records: Unpacked



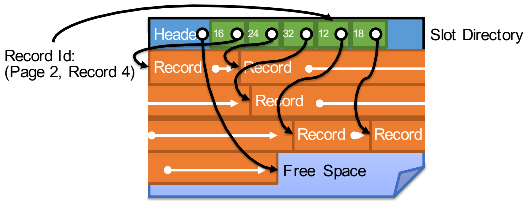
- Bitmap denotes "slots" with records
- Record id: record number in page
- **Insert:** find first empty slot
- **Delete:** Clear bit

Variable Length Records



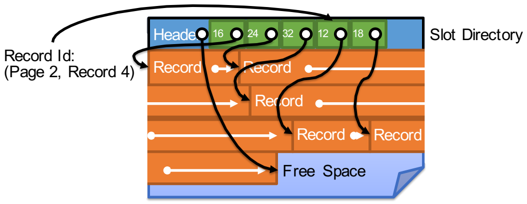
- How do we know where each record begins?
- What happens when we add and delete records?

Slotted Page



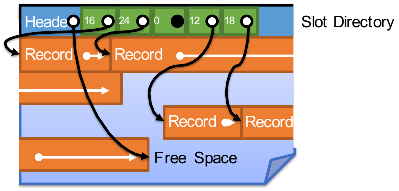
- Introduce **slot directory** in header
  - Length + Pointer to beginning of record
  - Pointer to free space
- Record ID = location in slot table
- ***Delete***? (e.g., 3rd record on the page)

Slotted Page: Delete 3rd record



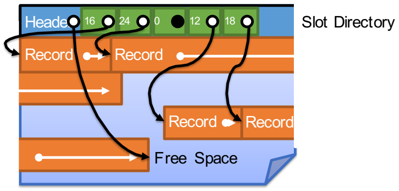
- **Delete**: Set pointer to null.
  - Doesn't affect pointers to other records
  - However, need to make sure we remove any references to **record_id** in indexes

Slotted Page



- ***Insert***?

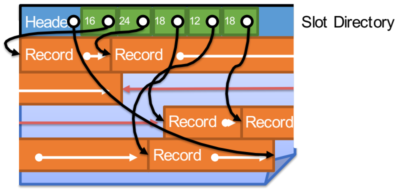
Slotted Page: Insert Record



- **Insert**:

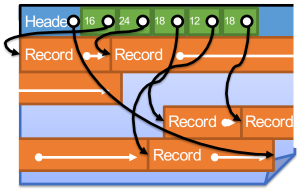
  - Place record in free space on page

Slotted Page



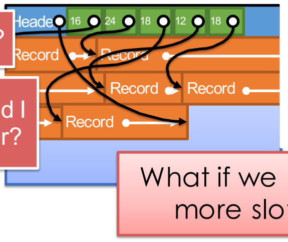
- **Insert**:
  - Place record in free space on page
  - Create pointer in next open slot in slot directory
  - Fragmentation?

Slotted Page



- **Insert**:
  - Place record in free space on page
  - Create pointer in next open slot in slot directory
  - Reorganize data on page.

Slotted Page



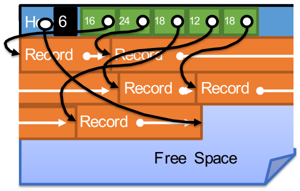
Is this safe?

When should I re-organizer?

What if we need more slots?

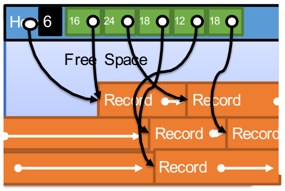
- **Insert**:
  - Place record in free space on page
  - Create pointer in next open slot in slot directory
  - Reorganize data on page.

Slotted Page: Growing Slots



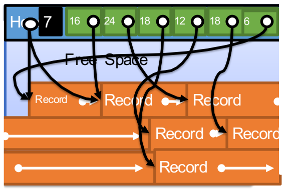
- Track number of slots in slot directory

Slotted Page: Growing Slots



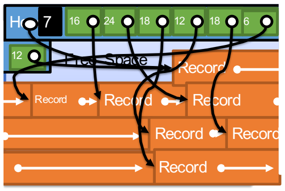
- Track number of slots in slot directory
- Grow records from other end of page
  - Why?

Slotted Page: Growing Slots

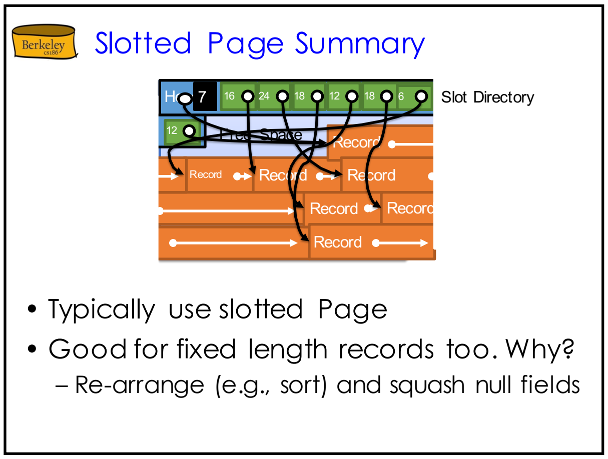


- Track number of slots in slot directory
- Grow records from other end of page
- Extend slot directory on insert
  - Add record in free space & update counter

Slotted Page: Growing Slots

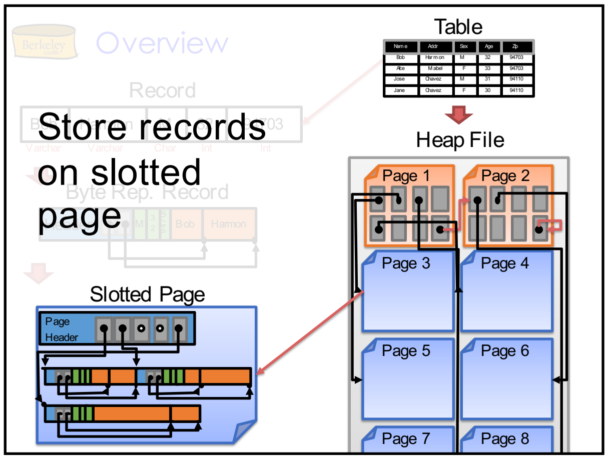


- Track number of slots in slot directory
- Grow records from other end of page
- Extend slot directory on insert
  - Add record in free space & update counter



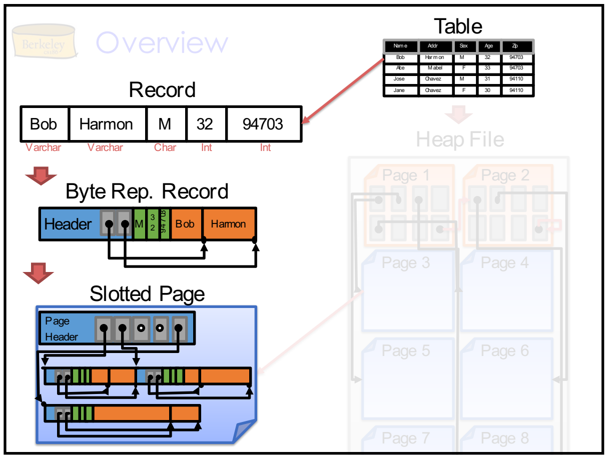
Slotted Page Summary

- Typically use slotted Page
- Good for fixed length records too. Why?
  - Re-arrange (e.g., sort) and squash null fields



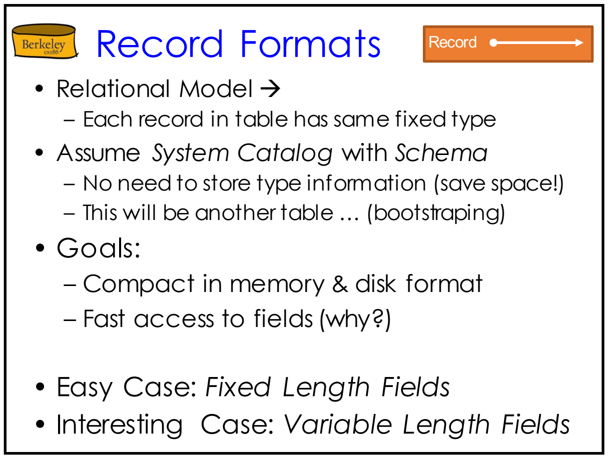
Overview

Store records on slotted page



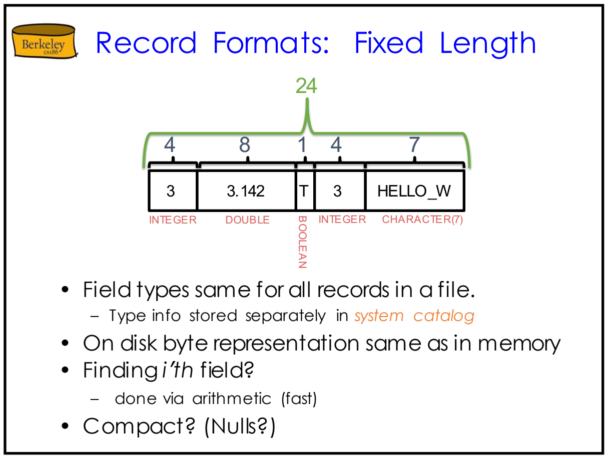
Overview

Bob	Harmon	M	32	94703
Varchar	Varchar	Char	Int	Int



Record Formats

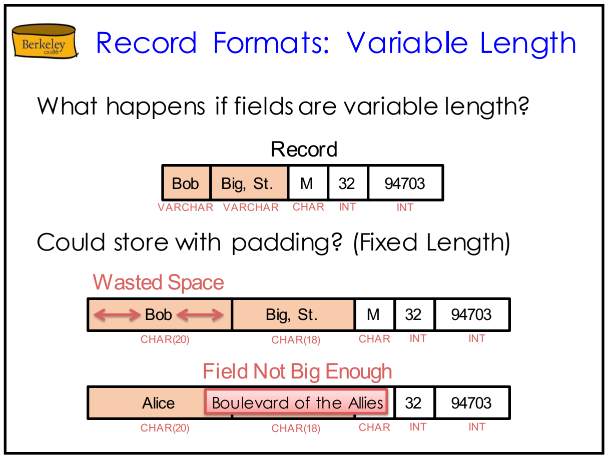
- Relational Model →
  - Each record in table has same fixed type
- Assume *System Catalog* with *Schema*
  - No need to store type information (save space!)
  - This will be another table … (bootstraping)
- Goals:
  - Compact in memory & disk format
  - Fast access to fields (why?)
- Easy Case: *Fixed Length Fields*
- Interesting Case: *Variable Length Fields*



Record Formats: Fixed Length

3	3.142	T	3	HELLO_W
INTEGER	DOUBLE	BOOLEAN	INTEGER	CHARACTER(7)

- Field types same for all records in a file.
  - Type info stored separately in *system catalog*
- On disk byte representation same as in memory
- Finding *i'th* field?
  - done via arithmetic (fast)
- Compact? (Nulls?)



Record Formats: Variable Length

What happens if fields are variable length?

Record

Bob	Big, St.	M	32	94703
VARCHAR	VARCHAR	CHAR	INT	INT

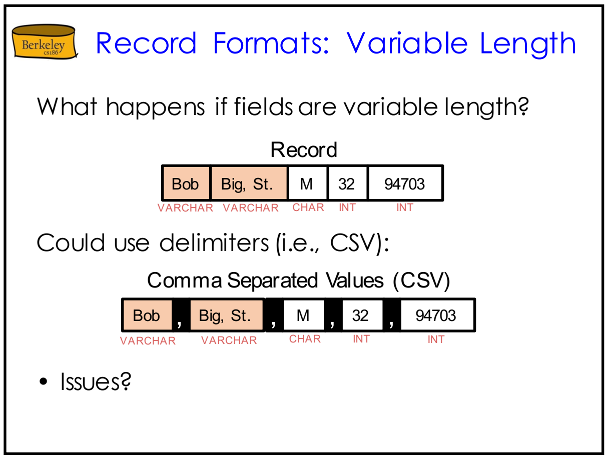
Could store with padding? (Fixed Length)

Wasted Space

Bob	Big, St.	M	32	94703
CHAR(20)	CHAR(18)	CHAR	INT	INT

Field Not Big Enough

Alice	Boulevard of the Allies	32	94703
CHAR(20)	CHAR(18)	CHAR	INT	INT



Record Formats: Variable Length

What happens if fields are variable length?

Record

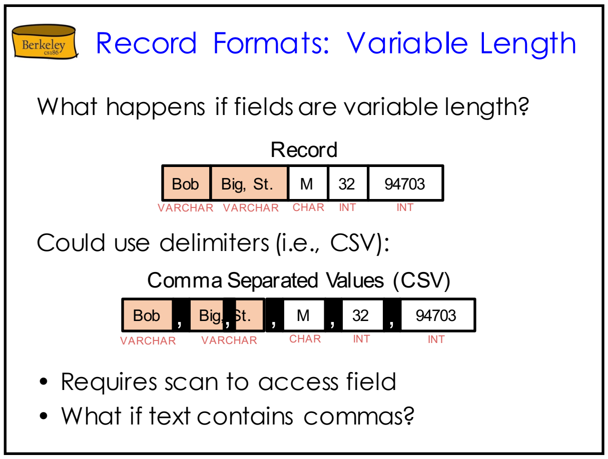
Bob	Big, St.	M	32	94703
VARCHAR	VARCHAR	CHAR	INT	INT

Could use delimiters (i.e., CSV):

Comma Separated Values (CSV)

Bob	,	Big, St.	,	M	,	32	,	94703
VARCHAR		VARCHAR		CHAR		INT		INT

- Issues?



Record Formats: Variable Length

What happens if fields are variable length?

Record

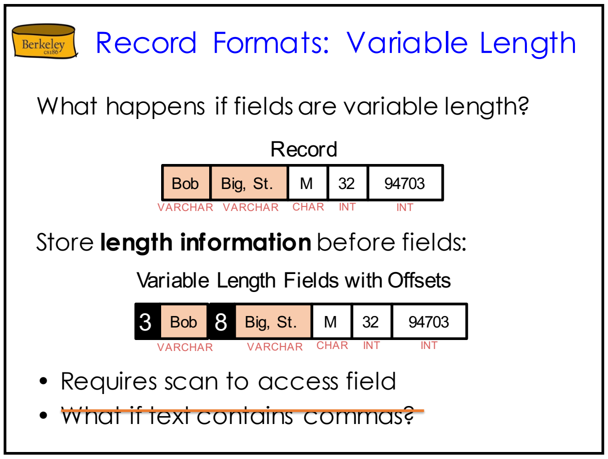
Bob	Big, St.	M	32	94703
VARCHAR	VARCHAR	CHAR	INT	INT

Could use delimiters (i.e., CSV):

Comma Separated Values (CSV)

Bob	,	Big, St.	,	M	,	32	,	94703
VARCHAR		VARCHAR		CHAR		INT		INT

- Requires scan to access field
- What if text contains commas?



Record Formats: Variable Length

What happens if fields are variable length?

Record

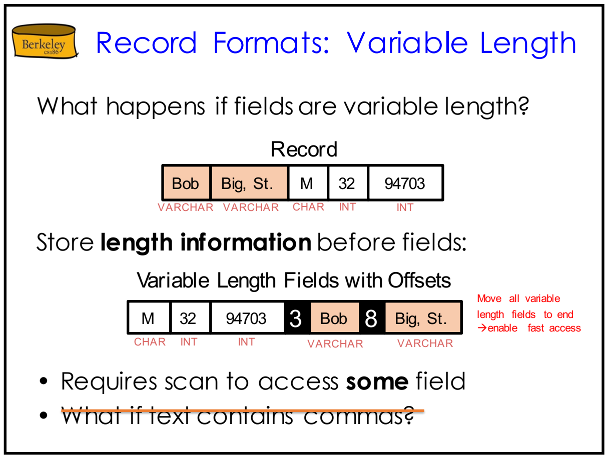
Bob	Big, St.	M	32	94703
VARCHAR	VARCHAR	CHAR	INT	INT

Store **length information** before fields:

Variable Length Fields with Offsets

3	Bob	8	Big, St.	M	32	94703
	VARCHAR		VARCHAR	CHAR	INT	INT

- Requires scan to access field
- ~~What if text contains commas?~~



Record Formats: Variable Length

What happens if fields are variable length?

Record

Bob	Big, St.	M	32	94703
VARCHAR	VARCHAR	CHAR	INT	INT

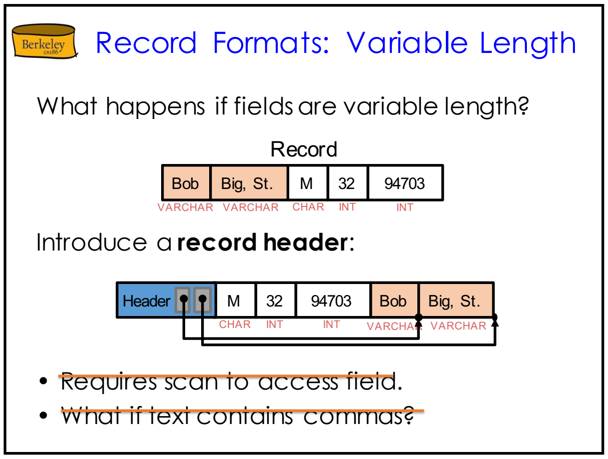
Store **length information** before fields:

Variable Length Fields with Offsets

M	32	94703	3	Bob	8	Big, St.
CHAR	INT	INT		VARCHAR		VARCHAR

Move all variable length fields to end
→enable fast access

- Requires scan to access **some** field
- ~~What if text contains commas?~~



Record Formats: Variable Length

What happens if fields are variable length?

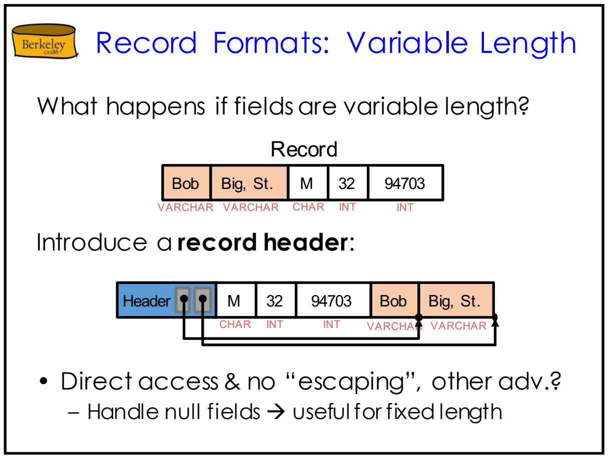
Record

Bob	Big, St.	M	32	94703
VARCHAR	VARCHAR	CHAR	INT	INT

Introduce a **record header**:

Header	●	●	M	32	94703	Bob	Big, St.
			CHAR	INT	INT	VARCHAR	VARCHAR

- ~~Requires scan to access field.~~
- ~~What if text contains commas?~~



Record Formats: Variable Length

What happens if fields are variable length?

Record

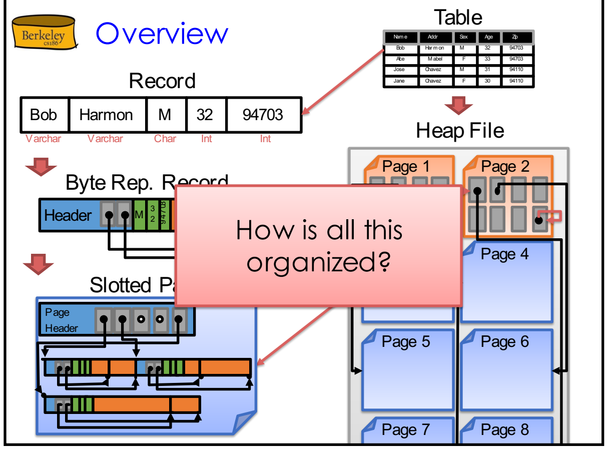
Bob	Big, St.	M	32	94703
VARCHAR	VARCHAR	CHAR	INT	INT

Introduce a **record header**:

Header	●	●	M	32	94703	Bob	Big, St.
			CHAR	INT	INT	VARCHAR	VARCHAR

- Direct access & no "escaping", other adv.?
  - Handle null fields → useful for fixed length

Overview



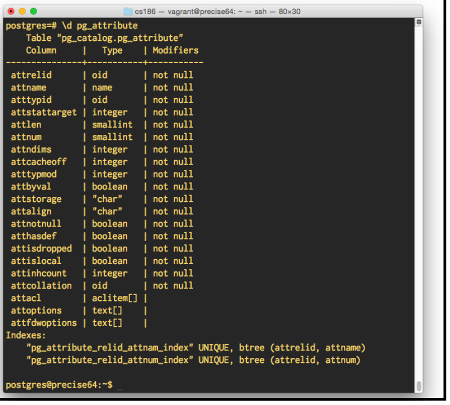
How is all this organized?

System Catalogs

- For each relation:
  - name, file location, file structure (e.g., Heap file)
  - attribute name and type, for each attribute
  - index name, for each index
  - integrity constraints
- For each index:
  - structure (e.g., B+ tree) and search key fields
- For each view:
  - view name and definition
- Plus statistics, authorization, buffer pool size, etc.

☛ *Catalogs are themselves stored as relations!*

pg_attribute

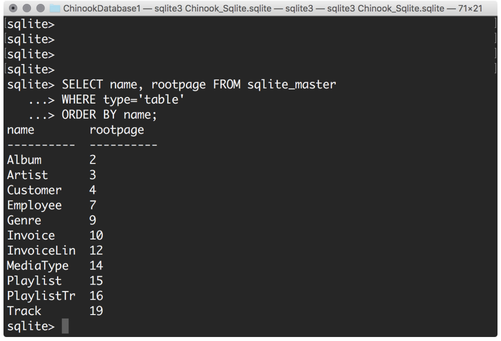


```
postgres=# \d pg_attribute
    Table "pg_catalog.pg_attribute"
    Column     |   Type    | Modifiers
---------------+-----------+-----------
 attrelid      | oid       | not null
 attname       | name      | not null
 atttypid      | oid       | not null
 attstattarget | integer   | not null
 attlen        | smallint  | not null
 attnum        | smallint  | not null
 attndims      | integer   | not null
 attcacheoff   | integer   | not null
 atttypmod     | integer   | not null
 attbyval      | boolean   | not null
 attstorage    | "char"    | not null
 attalign      | "char"    | not null
 attnotnull    | boolean   | not null
 atthasdef     | boolean   | not null
 attisdropped  | boolean   | not null
 attislocal    | boolean   | not null
 attinhcount   | integer   | not null
 attcollation  | oid       | not null
 attacl        | aclitem[] |
 attoptions    | text[]    |
 attfdwoptions | text[]    |
Indexes:
    "pg_attribute_relid_attnam_index" UNIQUE, btree (attrelid, attname)
    "pg_attribute_relid_attnum_index" UNIQUE, btree (attrelid, attnum)

postgres@precise64:~$ _
```

sqlite_master

```
SELECT name, rootpage FROM sqlite_master
WHERE type='table'
ORDER BY name;
```


ChinookDatabase1 — sqlite3 Chinook_Sqlite.sqlite — sqlite3 — sqlite3 Chinook_Sqlite.sqlite — 71×21

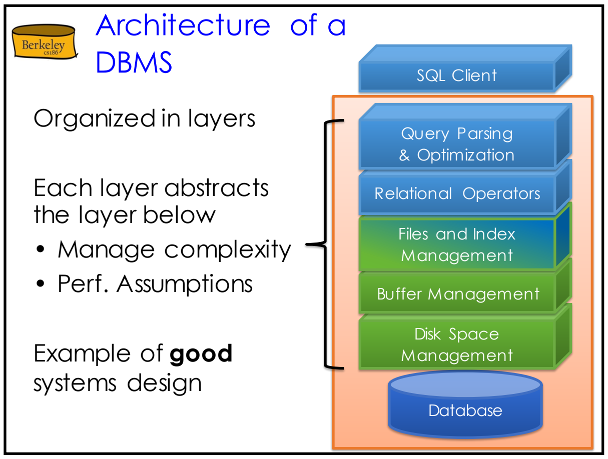
```
sqlite>
sqlite>
sqlite>
sqlite>
sqlite> SELECT name, rootpage FROM sqlite_master
   ...> WHERE type='table'
   ...> ORDER BY name;
name        rootpage
----------  ----------
Album       2
Artist      3
Customer    4
Employee    7
Genre       9
Invoice     10
InvoiceLin  12
MediaType   14
Playlist    15
PlaylistTr  16
Track       19
sqlite>
```

Summary

- Disk manager loads and stores pages
  - Block level reasoning
  - Abstracts device and file system; provides fast next
- Buffer manager brings pages into RAM
  - page pinned while reading/writing
  - dirty pages written to disk
  - good *replacement policy* essential for performance
- DBMS "File" tracks collection of pages, records within each.
  - Heap-files: unordered records organized with directories

Summary (Contd.)

- Slotted page format
  - Variable length records and intra-page reorg
- Variable length record format
  - Direct access to i'th field and null values.
- Catalog relations store information about relations, indexes and views.

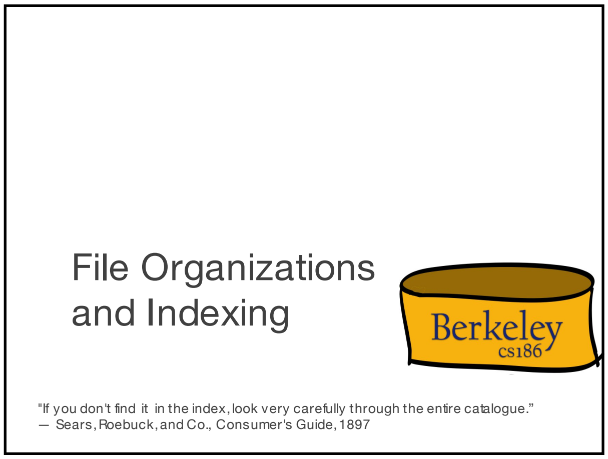


Architecture of a DBMS

Organized in layers

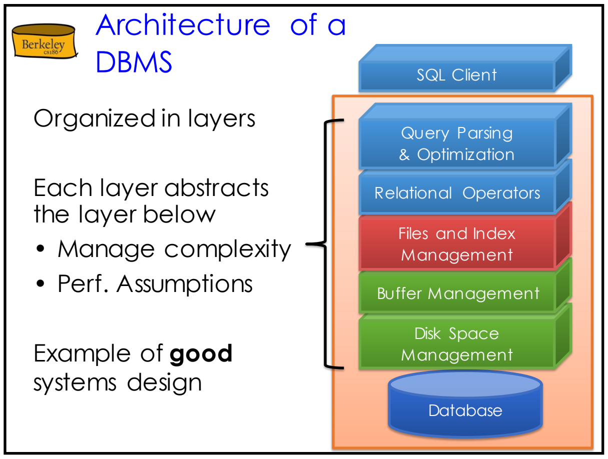
Each layer abstracts the layer below
- Manage complexity
- Perf. Assumptions

Example of **good** systems design



File Organizations and Indexing

"If you don't find it in the index, look very carefully through the entire catalogue."
— Sears, Roebuck, and Co., Consumer's Guide, 1897

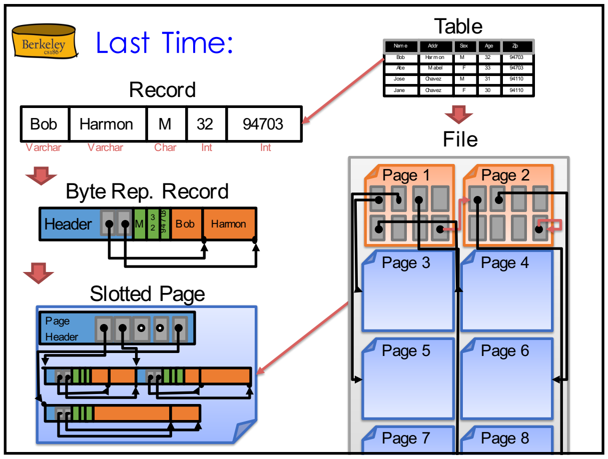


Architecture of a DBMS

Organized in layers

Each layer abstracts the layer below
- Manage complexity
- Perf. Assumptions

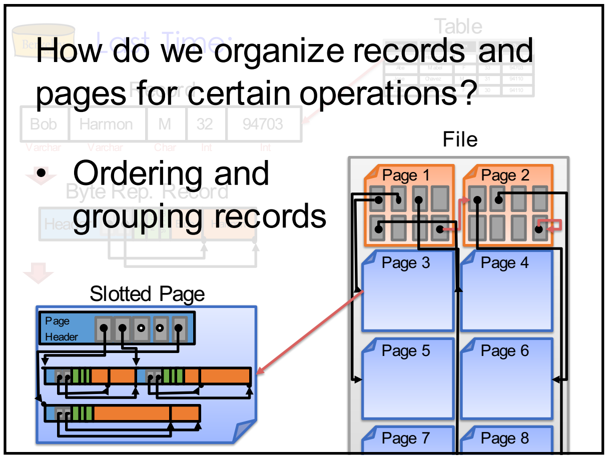
Example of **good** systems design



Last Time:

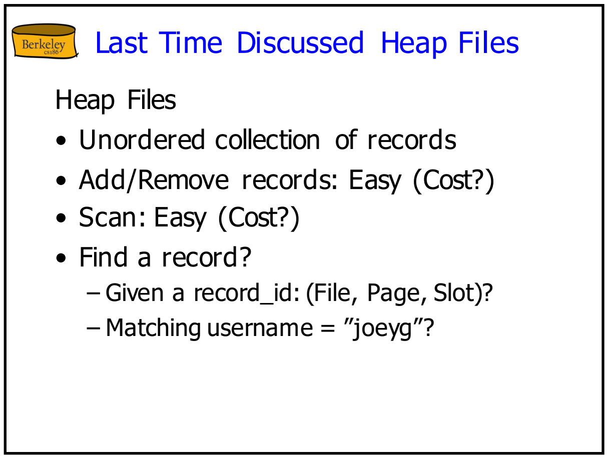
Table

Name	Addr	Sex	Age	Zip
Bob	Harmon	M	32	94703
Alce	Mabel	F	33	94703
Jose	Chavez	M	31	94110
Jane	Chavez	F	30	94110



How do we organize records and pages for certain operations?

- Ordering and grouping records



Last Time Discussed Heap Files

Heap Files
- Unordered collection of records
- Add/Remove records: Easy (Cost?)
- Scan: Easy (Cost?)
- Find a record?
  - Given a record_id: (File, Page, Slot)?
  - Matching username = ”joeyg”?

Multiple File Organizations

Many alternatives exist, *each good in some situations and not so good in others*:

- Heap files: Suitable when typical access is a full scan of all records
- Sorted Files: Best for retrieval in *search key* order, or a **range** of records is needed
- Clustered Files & Indexes: Group data into blocks to enable fast lookup and efficient modification. (More on this soon …)

Bigger Questions

- What is the "best" file organization?
  - Depends on access patterns…
  - how? what are they?
- Can we be quantitative about tradeoffs?
  - Better → How Much?

Goals for Today

- Big picture overheads for data access
  - We'll (overly) simplify things to **gain insight**
  - Still, a bit of discipline:
    - Clearly identify assumptions up front
    - Then estimate cost in a principled way
- Foundation for query optimization
  - Can't choose the fastest scheme without an estimate of speed!

Cost Model for Analysis

- **B:** The number of data blocks
- **R:** Number of records per block
- **D:** (Average) time to read or write disk block
- *Average-case* analyses for *uniform random* workloads
- We will ignore:
  - Sequential vs. Random I/O
  - Pre-fetching
  - Any in-memory costs

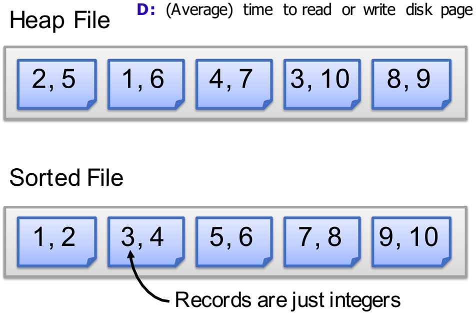
☛ ***Good enough to show the overall trends!***

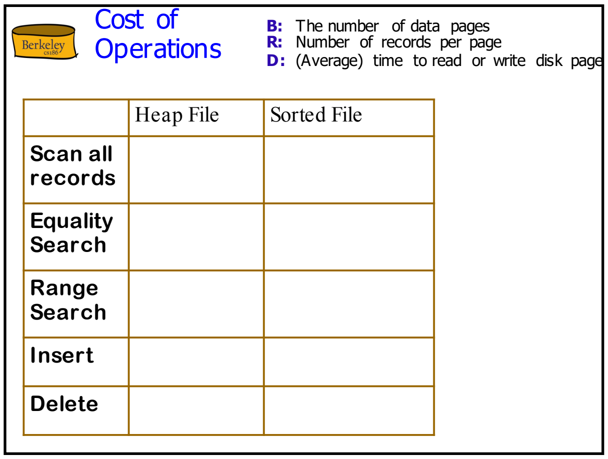
More Assumptions

- Single record insert and delete.
- Equality selection - exactly one match
- For Heap Files:
  - Insert always appends to end of file.
- For Sorted Files:
  - Files compacted after deletions.
  - Sorted according to search key.
- Question all these assumptions and rework
  - As an exercise to study for tests, generate ideas

Heap Files & Sorted Files

B: The number of data pages = 5
R: Number of records per page = 2
D: (Average) time to read or write disk page = 5 ms

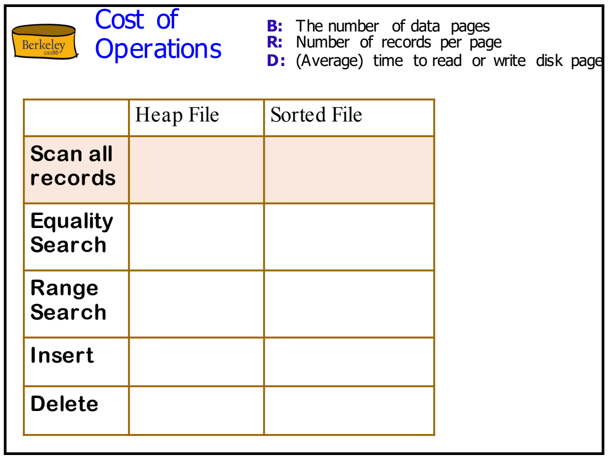




Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

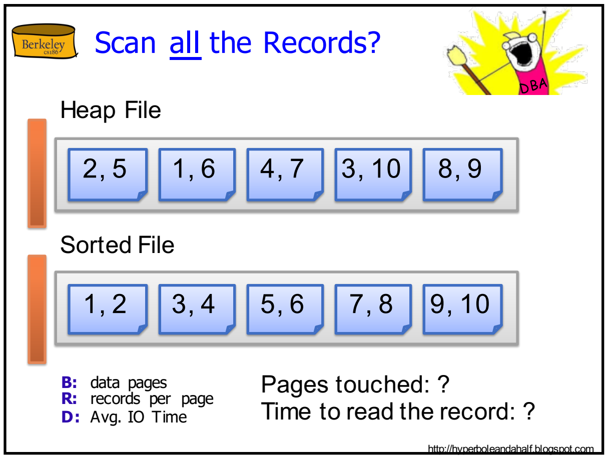
	Heap File	Sorted File
Scan all records		
Equality Search		
Range Search		
Insert		
Delete		



Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	Sorted File
Scan all records		
Equality Search		
Range Search		
Insert		
Delete		



Scan all the Records?

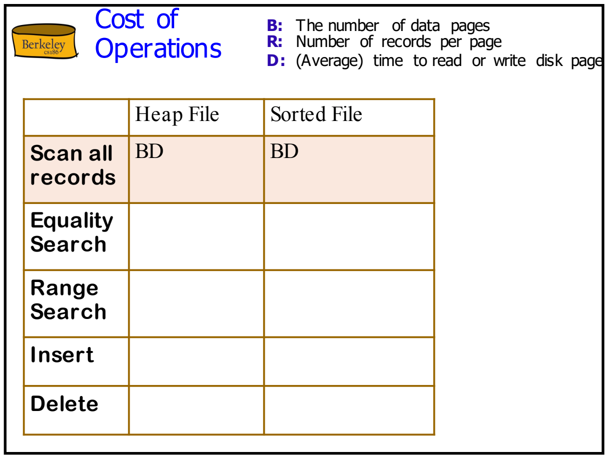
Heap File

Sorted File

B: data pages
R: records per page
D: Avg. IO Time

Pages touched: ?
Time to read the record: ?

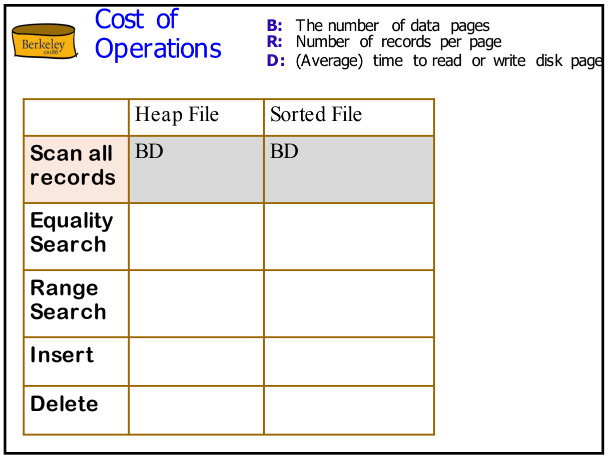
http://hyperboleandahalf.blogspot.com



Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

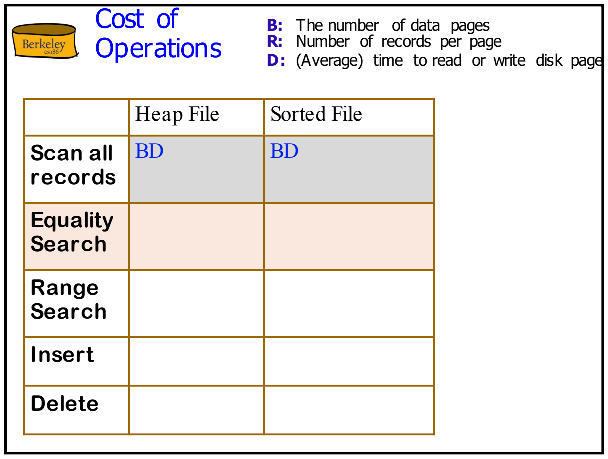
	Heap File	Sorted File
Scan all records	BD	BD
Equality Search		
Range Search		
Insert		
Delete		



Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	Sorted File
Scan all records	BD	BD
Equality Search		
Range Search		
Insert		
Delete		



Cost of Operations

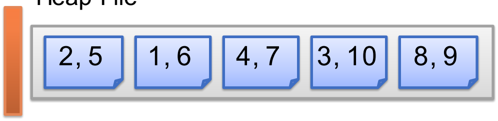
B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	Sorted File
Scan all records	BD	BD
Equality Search		
Range Search		
Insert		
Delete		

Find Key 8

B: data pages
R: records per page
D: Avg. IO Time

Heap File



Pages touched **on average**?

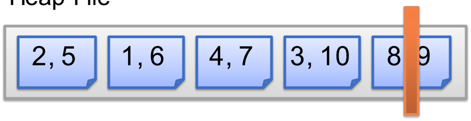
- **P**(i): Probability of key on page **i** is: **1/B**
- **T**(i): # Pages touched if key on page **i** is: **i**
- Therefore the expected # pages touched:

$$\sum_{i=1}^{B} \mathrm{T}(i)\mathbf{P}(i) = \sum_{i=1}^{B} i\frac{1}{B} = \frac{B(B+1)}{2B} \approx \frac{B}{2}$$

Find Key 8

B: data pages
R: records per page
D: Avg. IO Time

Heap File



Pages touched **on average**: B/2

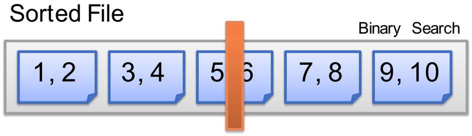
Breaking an Assumption:
What if there was more than one key?

- Need to check all pages → B

Find Key 8

B: data pages
R: records per page
D: Avg. IO Time

Sorted File



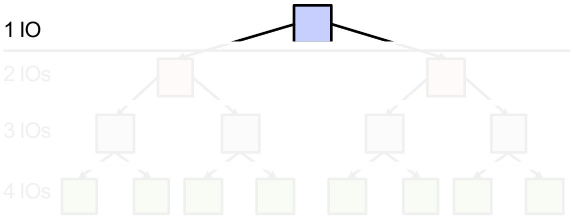
Worst-case: Pages touched in binary search

- **$\mathbf{Log_2 B}$**

Average-case: Pages touched in binary search

- **$\mathbf{Log_2 B}$?**

Average Case Binary Search



Expected Number of Reads: 1 (1 / B) + 2 (2 / B) + 3 (4 / B) + 4 (8 / B)

Average ≈ Worst

$$\sum_{i=1}^{\log_2 B} i\frac{2^{i-1}}{B} = \frac{1}{B}\sum_{i=1}^{\log_2 B} i2^{i-1} = \boxed{\log_2 B} - \frac{B-1}{B}$$

Cost of Operations

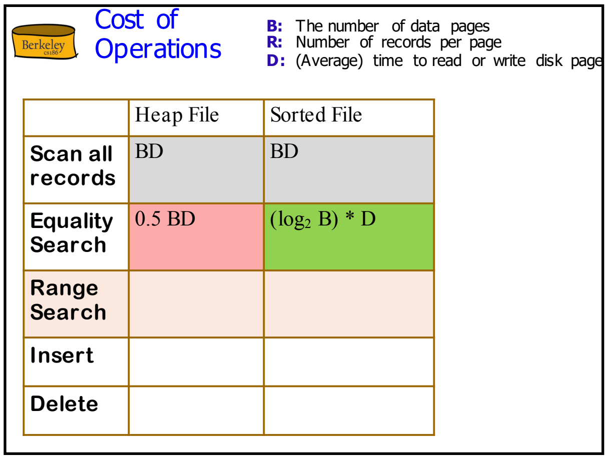
B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

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

Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

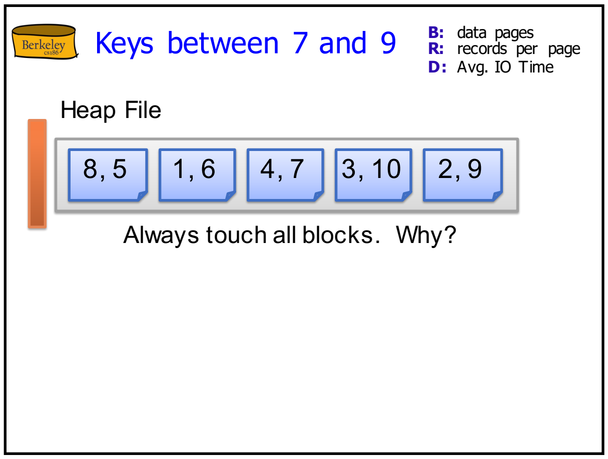
	Heap File	Sorted File
Scan all records	BD	BD
Equality Search	0.5 BD	($\log_2$ B) * D
Range Search		
Insert		
Delete		



Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

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



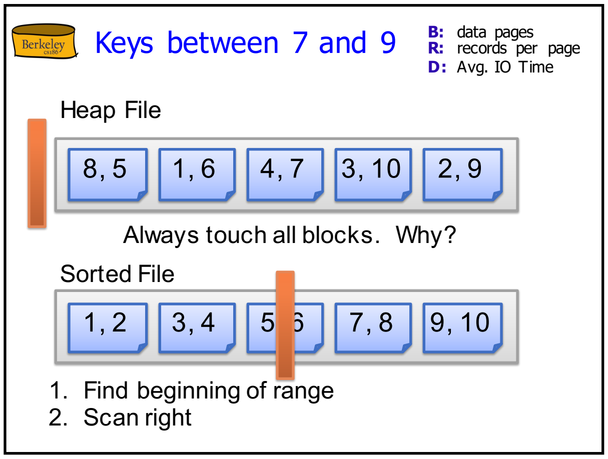
Keys between 7 and 9

B: data pages
R: records per page
D: Avg. IO Time

Heap File

8, 5 | 1, 6 | 4, 7 | 3, 10 | 2, 9

Always touch all blocks. Why?



Keys between 7 and 9

B: data pages
R: records per page
D: Avg. IO Time

Heap File

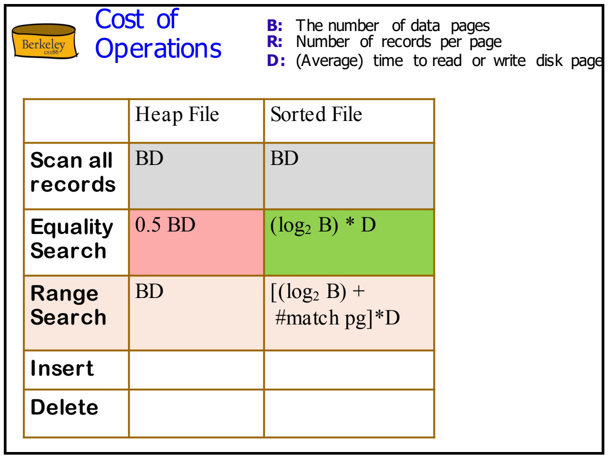
8, 5 | 1, 6 | 4, 7 | 3, 10 | 2, 9

Always touch all blocks. Why?

Sorted File

1, 2 | 3, 4 | 5, 6 | 7, 8 | 9, 10

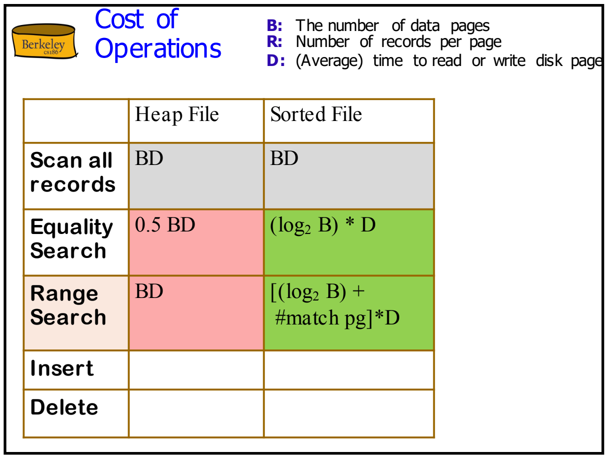
1. Find beginning of range
2. Scan right



Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

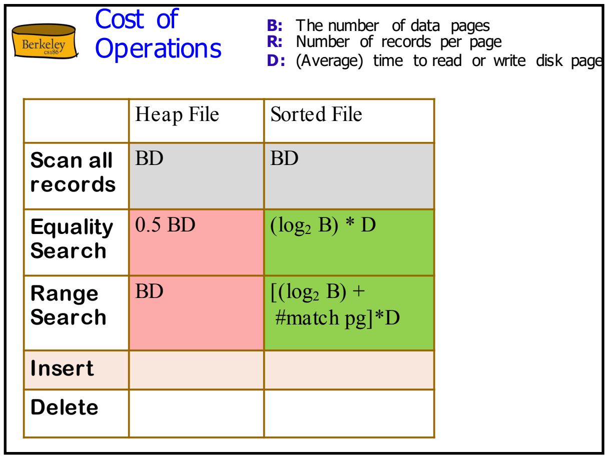
	Heap File	Sorted File
Scan all records	BD	BD
Equality Search	0.5 BD	$(\log_2 B) * D$
Range Search	BD	$[(\log_2 B) + \#\text{match pg}]*D$
Insert		
Delete		



Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

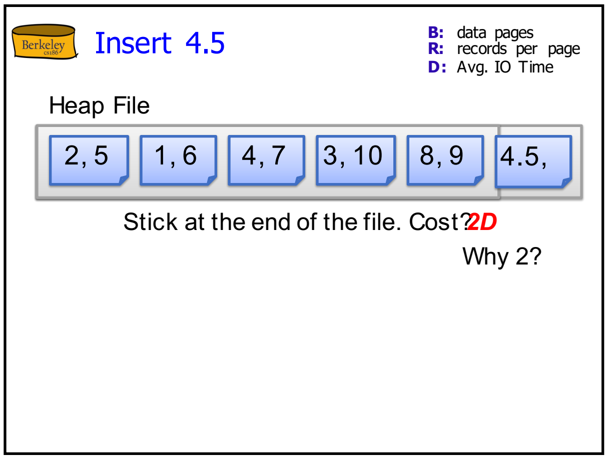
	Heap File	Sorted File
Scan all records	BD	BD
Equality Search	0.5 BD	$(\log_2 B) * D$
Range Search	BD	$[(\log_2 B) + \#\text{match pg}]*D$
Insert		
Delete		



Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	Sorted File
Scan all records	BD	BD
Equality Search	0.5 BD	$(\log_2 B) * D$
Range Search	BD	$[(\log_2 B) + \#\text{match pg}]*D$
Insert		
Delete		



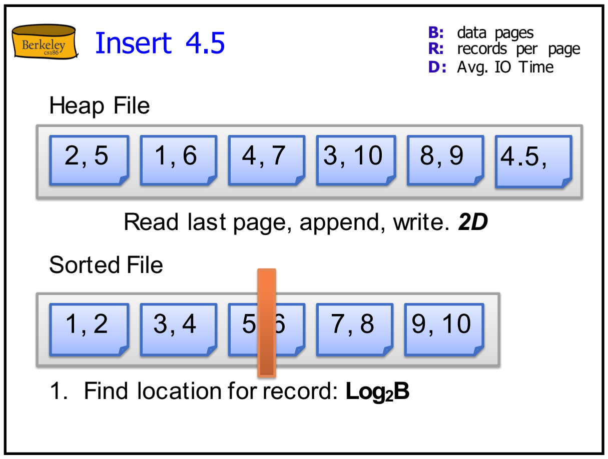
Insert 4.5

B: data pages
R: records per page
D: Avg. IO Time

Heap File

Stick at the end of the file. Cost? *2D*

Why 2?



Insert 4.5

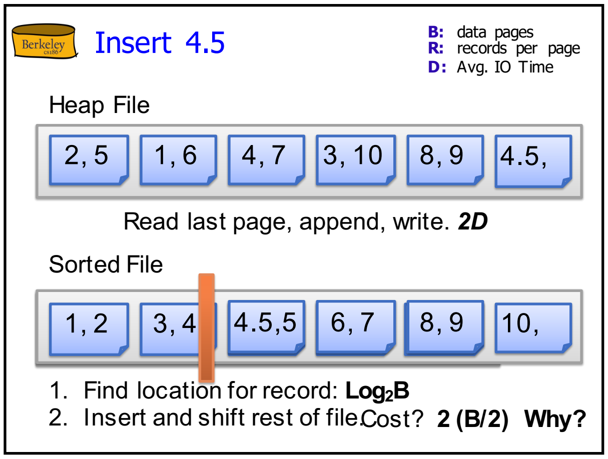
B: data pages
R: records per page
D: Avg. IO Time

Heap File

Read last page, append, write. ***2D***

Sorted File

1. Find location for record: **Log_2B**



Insert 4.5

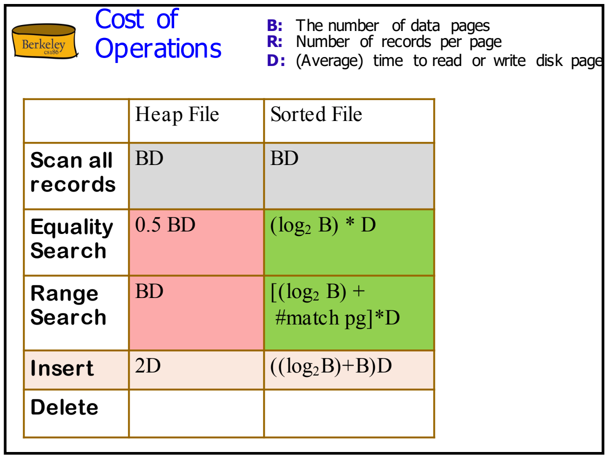
B: data pages
R: records per page
D: Avg. IO Time

Heap File

Read last page, append, write. ***2D***

Sorted File

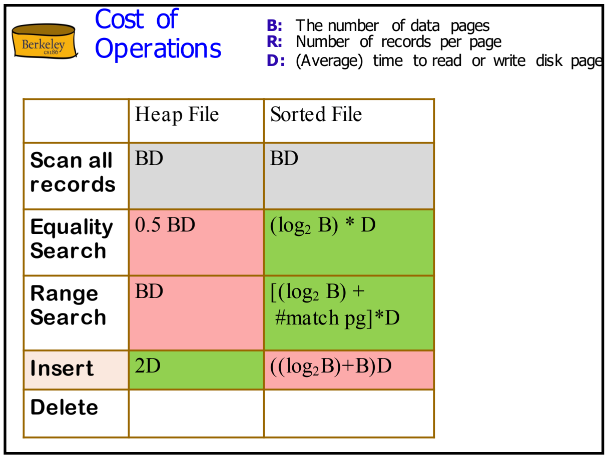
1. Find location for record: **Log_2B**
2. Insert and shift rest of file Cost? **2 (B/2) Why?**



Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

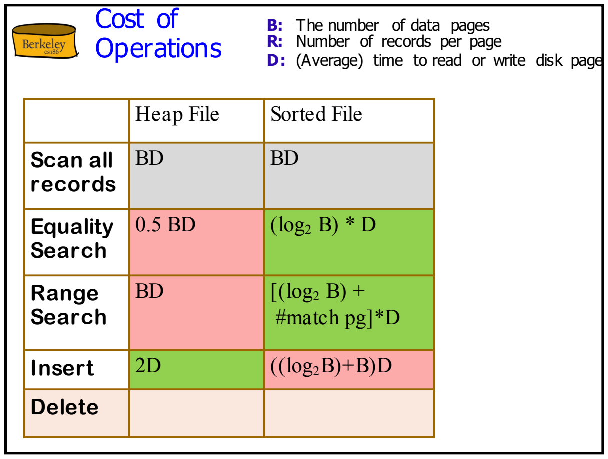
	Heap File	Sorted File
Scan all records	BD	BD
Equality Search	0.5 BD	$(\log_2 B) * D$
Range Search	BD	$[(\log_2 B) +$ #match pg]*D
Insert	2D	$((\log_2 B)+B)D$
Delete		



Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

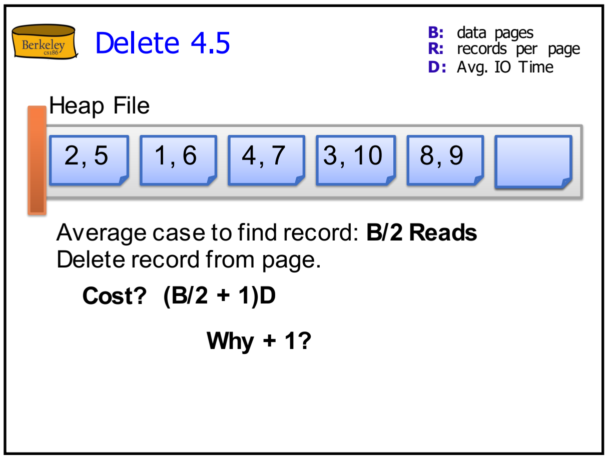
	Heap File	Sorted File
Scan all records	BD	BD
Equality Search	0.5 BD	$(\log_2 B) * D$
Range Search	BD	$[(\log_2 B) +$ #match pg]*D
Insert	2D	$((\log_2 B)+B)D$
Delete		



Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	Sorted File
Scan all records	BD	BD
Equality Search	0.5 BD	$(\log_2 B) * D$
Range Search	BD	$[(\log_2 B) +$ #match pg]*D
Insert	2D	$((\log_2 B)+B)D$
Delete		



Delete 4.5

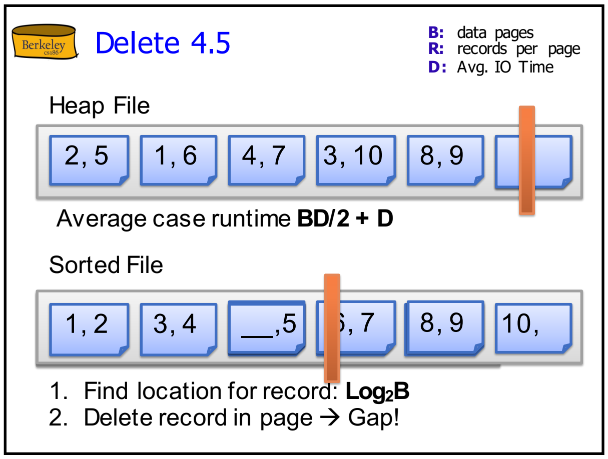
B: data pages
R: records per page
D: Avg. IO Time

Heap File

Average case to find record: **B/2 Reads**
Delete record from page.

Cost? (B/2 + 1)D

Why + 1?



Delete 4.5

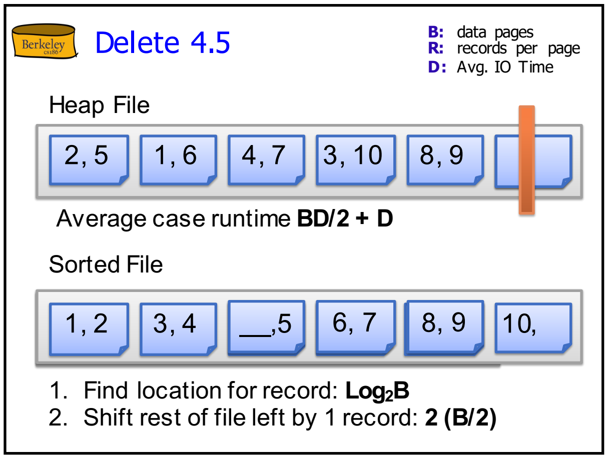
B: data pages
R: records per page
D: Avg. IO Time

Heap File

Average case runtime **BD/2 + D**

Sorted File

1. Find location for record: **Log$_2$B**
2. Delete record in page → Gap!



Delete 4.5

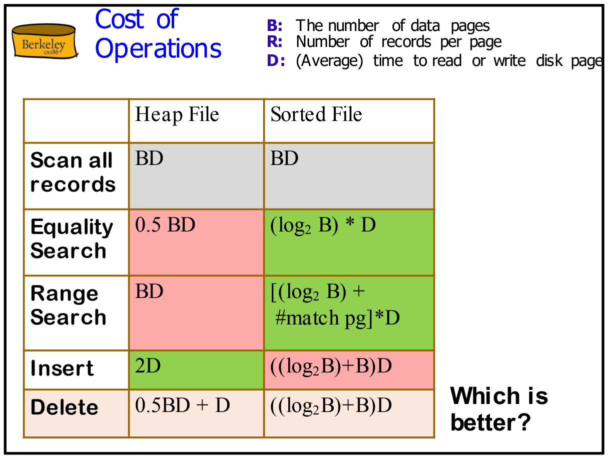
B: data pages
R: records per page
D: Avg. IO Time

Heap File

Average case runtime **BD/2 + D**

Sorted File

1. Find location for record: **Log$_2$B**
2. Shift rest of file left by 1 record: **2 (B/2)**

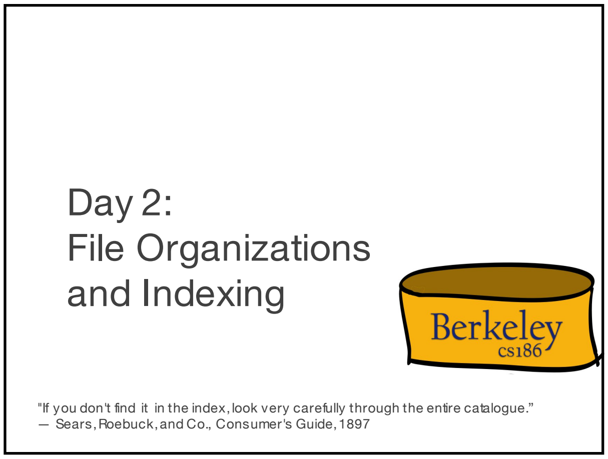


Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

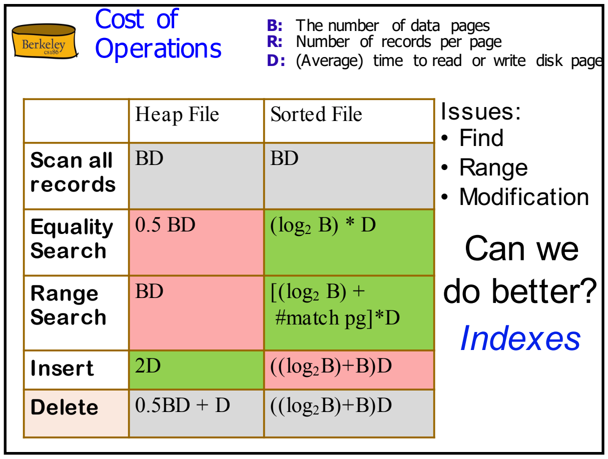
	Heap File	Sorted File
Scan all records	BD	BD
Equality Search	0.5 BD	$(\log_2 B) * D$
Range Search	BD	$[(\log_2 B) +$ #match pg$]*D$
Insert	2D	$((\log_2 B)+B)D$
Delete	0.5BD + D	$((\log_2 B)+B)D$

Which is better?



Day 2:
File Organizations
and Indexing

"If you don't find it in the index, look very carefully through the entire catalogue."
— Sears, Roebuck, and Co., Consumer's Guide, 1897



Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	Sorted File
Scan all records	BD	BD
Equality Search	0.5 BD	$(\log_2 B) * D$
Range Search	BD	$[(\log_2 B) +$ #match pg$]*D$
Insert	2D	$((\log_2 B)+B)D$
Delete	0.5BD + D	$((\log_2 B)+B)D$

Issues:
- Find
- Range
- Modification

Can we do better?

Indexes

Overview: Indexes

*An **index** is a data structure that enables fast lookup of data entries by search key*

- **Lookup:** may support many different operations
  - Equivalence, range, region, …
- **Search Key:** any *subset* of columns in the relation
  - Do not need to be unique
- **Data Entries:** items stored in the index, (**k**, {items})
  - Could be actual records or record ids (pointers)…
- **Many Types:** B-Tree, Hash, ISAM, R-Tree, GiST, …
- **Modification:** often support fast insert and delete
  - Lazily maintain ordering, clustering, …

Kinds of Lookups Supported?

- Basic Selection: <key> <op> <constant>
  - Equality selections (op is =)?
  - Range selections (op is one of <, >, <=, >=, BETWEEN)
- More exotic selections:
  - 2-dimensional ranges ("east of Berkeley and west of Truckee and North of Fresno and South of Eureka")
  - 2-dimensional radii ("within 2 miles of Soda Hall")
  - Common **n-dimensional index**: *R-tree, KD-Tree*
    - Beware of the ***curse of dimensionality***
  - Ranking queries ("10 restaurants closest to Berkeley")
  - Regular expression matches, genome string matches, etc.
  - See http://en.wikipedia.org/wiki/GiST for more

Search Key: *Any* Subset of Columns

- Search key needn't be a key of the relation
  - Recall: key of a relation must be unique (e.g., SSN)
  - Search keys don't have to be unique
- **Composite Keys**: more than one column
  - Think: Phone Book <Last Name, First>
  - **Lexicographic order**
  - <Age, Salary>:
    - ✓ Age = 31 & Sal = 400
    - ✓ Age = 55 & Sal > 200
    - ✗ Age > 31 & Sal = 400
    - ✓ Age = 31
    - ✓ Age > 31
    - ✗ Sal = 300

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

✗ Means that the index is unable able to exclude all entries that are not in the result set.

Data Entries: How are they stored?

- What is the representation of data in the index?
  - Actual data or pointer(s) to the data
- How is the data stored in the data file?
  - Clustered or unclusted with respect to the indexed
- Big Impact on Performance

How is data stored in the index

- Three alternatives:
  1. **By Value:** actual data record (with key value **k**)
  2. **By Reference:** <**k**, rid of matching data record>
  3. **By List of Refs.:** <**k**, list of rids of *all* matching data records>
- Choice is orthogonal to the indexing technique.
  - B+ trees, hash-based structures, R trees, GiSTs, …
- 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*.

Alternatives for Data Entries (Contd.)

Alternative 1 (By Value):
Actual data record (with key value ***k***)

- Index as a file organization for records
  - Alongside heap files or sorted files
- No "**pointer lookups**" to get data records
  - Following record ids
- Could a single relation have multiple indexes of this form?
  - Probably but it would be a bad idea. Why?

Alternatives for Data Entries (Contd.)

Alternative 2 (By Reference)

<**k**, rid of matching data record>

and Alternative 3 (By List of References)

<**k**, list of rids of matching data records>

Alternative 2

Key	Record Id
Gonzalez	1
Gonzalez	2
Gonzalez	3
Hong	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

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

- Alts. 2 or 3 needed to support multiple indexes per relation!
- Alt. 3 more compact than Alt. 2
- For very large rid lists, single data entry spans multiple blocks

Clustered vs. Unclustered Index

- In a clustered index:
  - index data entries are stored in (approximate) order by value of search keys in data records

Clustered vs. Unclustered Index

- In a clustered index:
  - index data entries are stored in (approximate) order by value of search keys in data records

Clustered

Key	Record Id
Gonzalez	1
Gonzalez	2
Gonzalez	3
Hong	4

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

Unclustered

Key	Record Id
Gonzalez	3
Hong	4
Gonzalez	1
Gonzalez	2

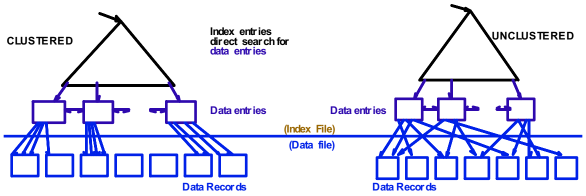
Clustered vs. Unclustered Index

- In a clustered index:
  - index data entries are stored in (approximate) order by value of search keys in data records
  - A file can be clustered on at most one search key.
  - Cost of retrieving data records through index varies *greatly* based on whether index is clustered or not!
  - Can Alternative 1 be un-clustered?
    - No, but clustered does not imply alt. 1 (earlier figure)
- Note: there is another definition of "clustering"
  - **Data Mining/AI**: grouping similar items in n-space

Clustered vs. Unclustered Index

Alternative 2 (by ref) data entries, data records in a Heap file.

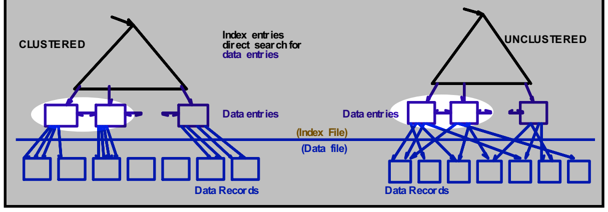
- To build clustered index, first sort the heap file
  - Leave some free space on each block for future inserts
- Overflow blocks may be needed for inserts.
  - Thus, order of data recs is "close to", but not identical to, the sort order.



Clustered vs. Unclustered Index

Alternative 2 (by ref) data entries, data records in a Heap file.

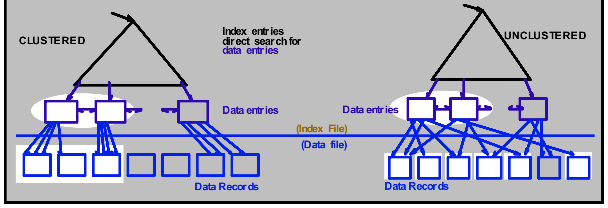
- To build clustered index, first sort the Heap file
  - Leave some free space on each block for future inserts
- Overflow blocks may be needed for inserts.
  - Thus, order of data recs is "close to", but not identical to, the sort order.



Clustered vs. Unclustered Index

Alternative 2 (by ref) data entries, data records in a Heap file.

- To build clustered index, first sort the Heap file
  - Leave some free space on each block for future inserts
- Overflow blocks may be needed for inserts.
  - Thus, order of data recs is "close to", but not identical to, the sort order.



Unclustered vs. Clustered Indexes

- Clustered Pros
  - Efficient for range searches
  - Potential locality benefits?
    - Sequential disk access, prefetching, etc.
  - Support certain types of compression
    - More soon on this topic
- Clustered Cons
  - More expensive to maintain
    - Need to update index data structure
    - Solution: on the fly or "lazily" via reorgs
  - Heap file usually only **packed to 2/3** to accommodate inserts

Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	Sorted File
Scan all records	BD	BD
Equality Search	0.5 BD	($\log_2$ B) * D
Range Search	BD	[($\log_2$ B) + #match pg]*D
Insert	2D	(($\log_2$B)+B)D
Delete	0.5BD + D	(($\log_2$B)+B)D

Issues:

- Find
- Range
- Modification

Can we do better?

Indexes

Cost of Operations

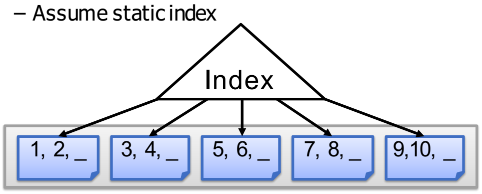
B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	Sorted File	Clustered Index
Scan all records	BD	BD	
Equality Search	0.5 BD	($\log_2$ B) * D	
Range Search	BD	[($\log_2$ B) + #match pg]*D	
Insert	2D	(($\log_2$B)+B)D	
Delete	0.5BD + D	(($\log_2$B)+B)D	

Simple Clustered Index Analysis

Assumptions:

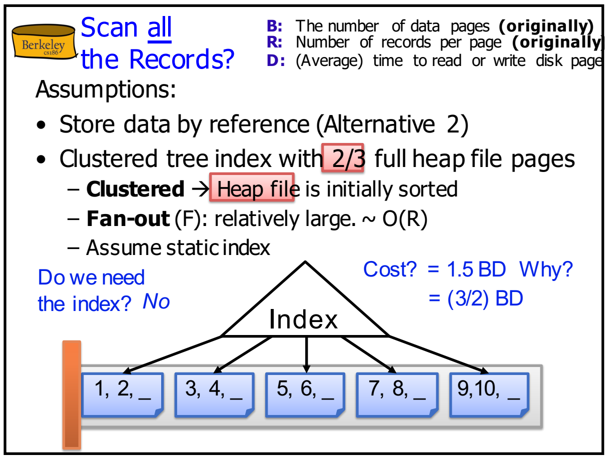
- Store data by reference (Alternative 2)
- Clustered tree index with 2/3 full heap file pages
  - Clustered → Heap file is initially sorted
  - **Fan-out** (F): relatively large. Why?
    - Page of <key, pointer> pairs ~ O(R)
  - Assume static index



Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	Sorted File	Clustered Index
Scan all records	BD	BD	?
Equality Search	0.5 BD	($\log_2$ B) * D	
Range Search	BD	[($\log_2$ B) + #match pg]*D	
Insert	2D	(($\log_2$B)+B)D	
Delete	0.5BD + D	(($\log_2$B)+B)D	



Scan all the Records?

B: The number of data pages **(originally)**
R: Number of records per page **(originally)**
D: (Average) time to read or write disk page

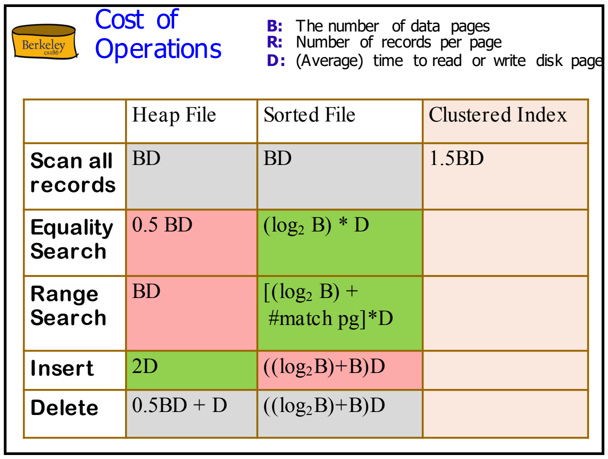
Assumptions:

- Store data by reference (Alternative 2)
- Clustered tree index with 2/3 full heap file pages
  - **Clustered** → Heap file is initially sorted
  - **Fan-out** (F): relatively large. ~ O(R)
  - Assume static index

Do we need the index? *No*

Cost? = 1.5 BD Why?
= (3/2) BD

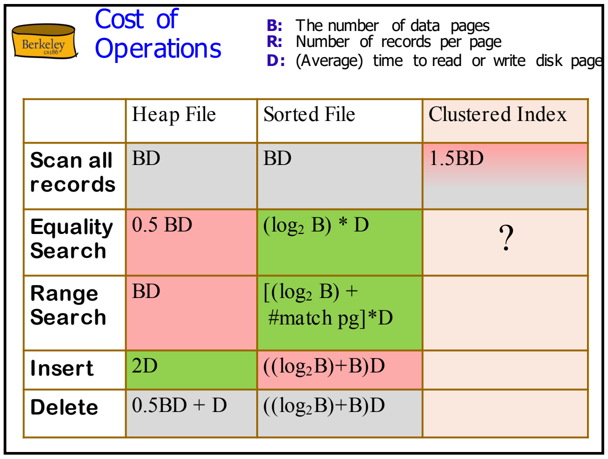
Index
1, 2, _
3, 4, _
5, 6, _
7, 8, _
9,10, _



Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

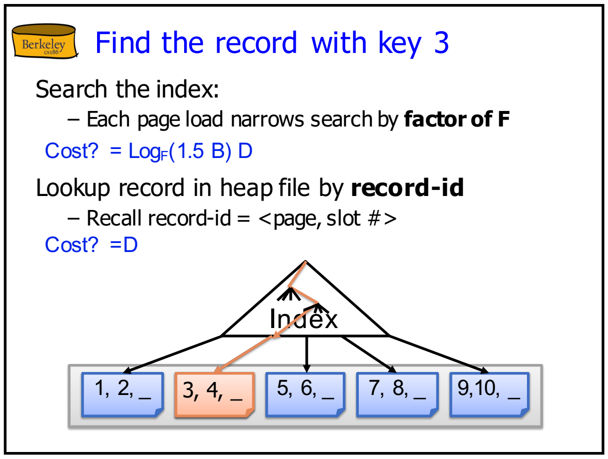
	Heap File	Sorted File	Clustered Index
Scan all records	BD	BD	1.5BD
Equality Search	0.5 BD	$(\log_2 B) * D$	
Range Search	BD	$[(\log_2 B) +$ #match pg$]*D$	
Insert	2D	$((\log_2 B)+B)D$	
Delete	0.5BD + D	$((\log_2 B)+B)D$	



Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	Sorted File	Clustered Index
Scan all records	BD	BD	1.5BD
Equality Search	0.5 BD	$(\log_2 B) * D$	?
Range Search	BD	$[(\log_2 B) +$ #match pg$]*D$	
Insert	2D	$((\log_2 B)+B)D$	
Delete	0.5BD + D	$((\log_2 B)+B)D$	



Find the record with key 3

Search the index:

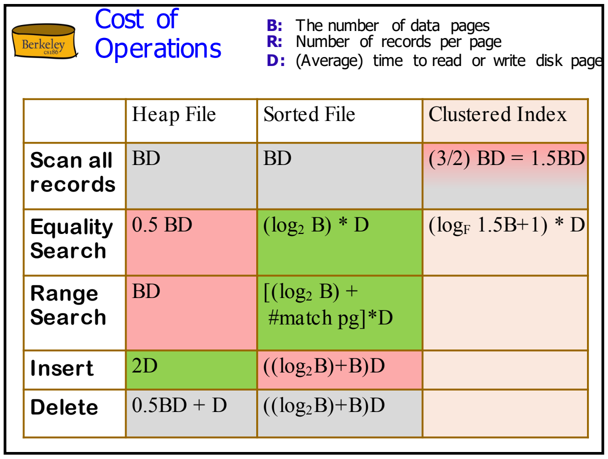
- Each page load narrows search by **factor of F**

Cost? = $\log_F(1.5\ B)\ D$

Lookup record in heap file by **record-id**

- Recall record-id = <page, slot #>

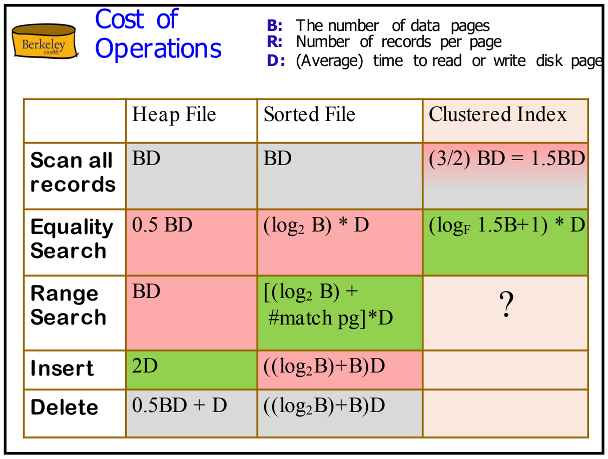
Cost? =D



Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

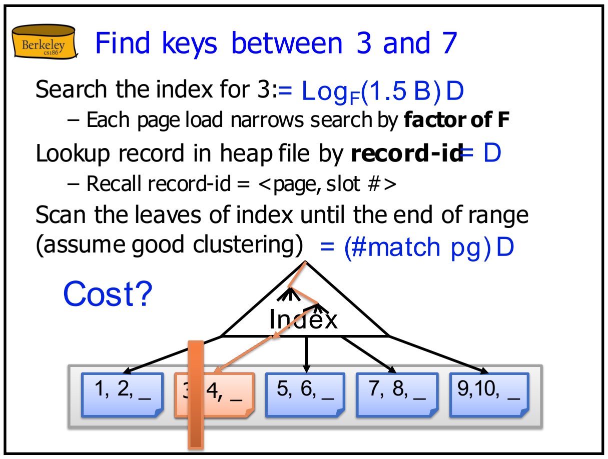
	Heap File	Sorted File	Clustered Index
Scan all records	BD	BD	(3/2) BD = 1.5BD
Equality Search	0.5 BD	$(\log_2 B) * D$	$(\log_F 1.5B+1) * D$
Range Search	BD	$[(\log_2 B) +$ #match pg$]*D$	
Insert	2D	$((\log_2 B)+B)D$	
Delete	0.5BD + D	$((\log_2 B)+B)D$	



Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	Sorted File	Clustered Index
Scan all records	BD	BD	(3/2) BD = 1.5BD
Equality Search	0.5 BD	$(\log_2 B) * D$	$(\log_F 1.5B+1) * D$
Range Search	BD	$[(\log_2 B) +$ #match pg$]*D$	?
Insert	2D	$((\log_2 B)+B)D$	
Delete	0.5BD + D	$((\log_2 B)+B)D$	



Find keys between 3 and 7

Search the index for 3:= $\text{Log}_F(1.5\,B)\,D$

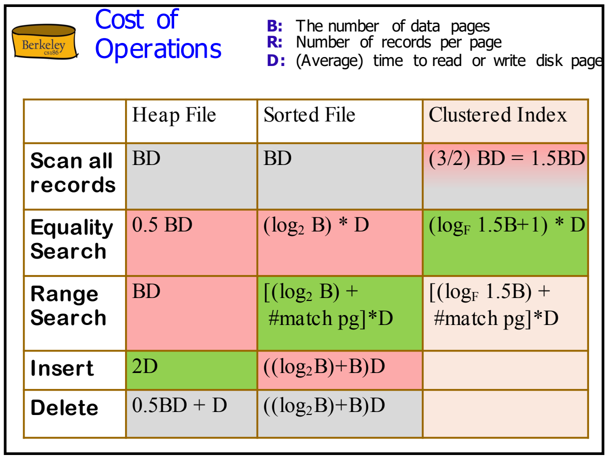
– Each page load narrows search by **factor of F**

Lookup record in heap file by **record-id**= D

– Recall record-id = <page, slot #>

Scan the leaves of index until the end of range (assume good clustering) = (#match pg) D

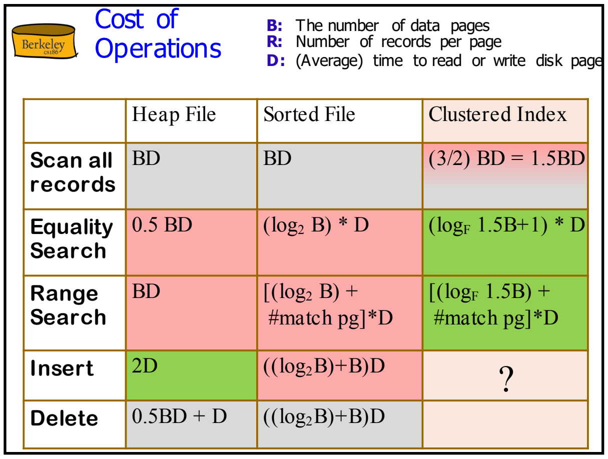
Cost?



Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

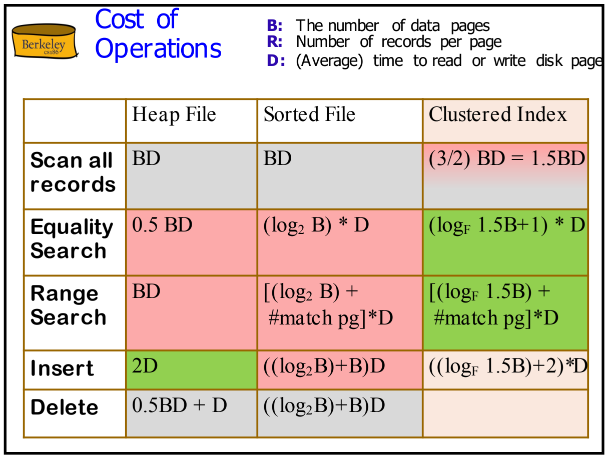
	Heap File	Sorted File	Clustered Index
Scan all records	BD	BD	(3/2) BD = 1.5BD
Equality Search	0.5 BD	$(\log_2 B) * D$	$(\log_F 1.5B+1) * D$
Range Search	BD	$[(\log_2 B) +$ #match pg]*D	$[(\log_F 1.5B) +$ #match pg]*D
Insert	2D	$((\log_2 B)+B)D$	
Delete	0.5BD + D	$((\log_2 B)+B)D$	



Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

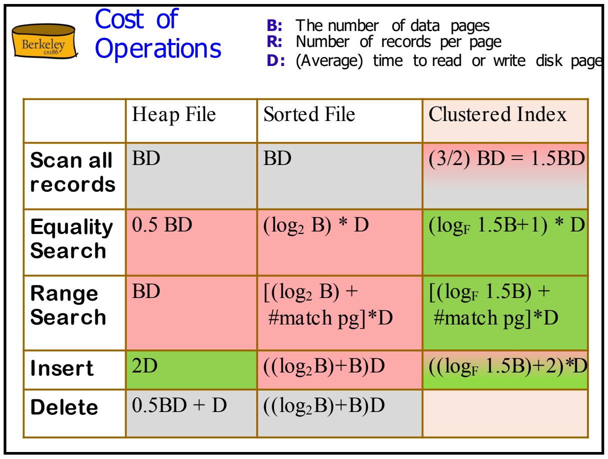
	Heap File	Sorted File	Clustered Index
Scan all records	BD	BD	(3/2) BD = 1.5BD
Equality Search	0.5 BD	$(\log_2 B) * D$	$(\log_F 1.5B+1) * D$
Range Search	BD	$[(\log_2 B) +$ #match pg]*D	$[(\log_F 1.5B) +$ #match pg]*D
Insert	2D	$((\log_2 B)+B)D$	?
Delete	0.5BD + D	$((\log_2 B)+B)D$	



Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

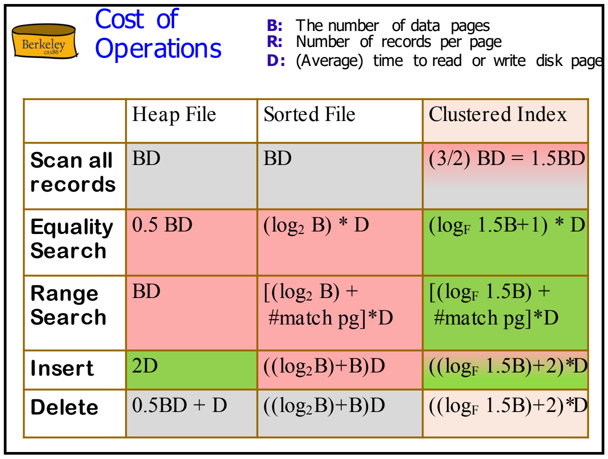
	Heap File	Sorted File	Clustered Index
Scan all records	BD	BD	(3/2) BD = 1.5BD
Equality Search	0.5 BD	$(\log_2 B) * D$	$(\log_F 1.5B+1) * D$
Range Search	BD	$[(\log_2 B) +$ #match pg]*D	$[(\log_F 1.5B) +$ #match pg]*D
Insert	2D	$((\log_2 B)+B)D$	$((\log_F 1.5B)+2)*D$
Delete	0.5BD + D	$((\log_2 B)+B)D$	



Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

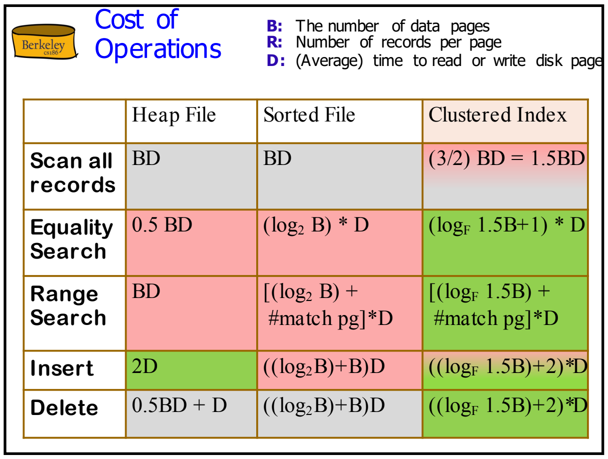
	Heap File	Sorted File	Clustered Index
Scan all records	BD	BD	(3/2) BD = 1.5BD
Equality Search	0.5 BD	$(\log_2 B) * D$	$(\log_F 1.5B+1) * D$
Range Search	BD	$[(\log_2 B) +$ #match pg]*D	$[(\log_F 1.5B) +$ #match pg]*D
Insert	2D	$((\log_2 B)+B)D$	$((\log_F 1.5B)+2)*D$
Delete	0.5BD + D	$((\log_2 B)+B)D$	



Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

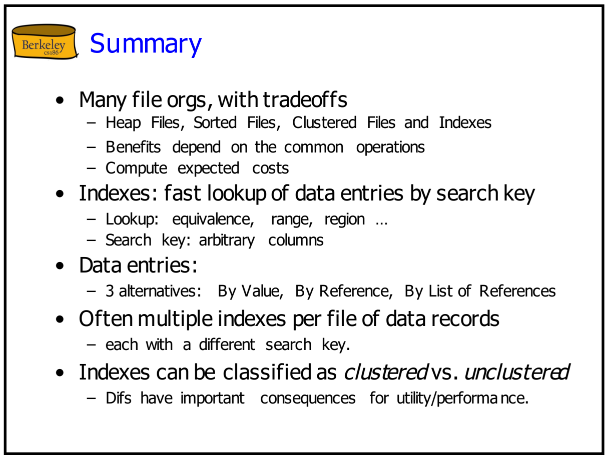
	Heap File	Sorted File	Clustered Index
Scan all records	BD	BD	(3/2) BD = 1.5BD
Equality Search	0.5 BD	$(\log_2 B) * D$	$(\log_F 1.5B+1) * D$
Range Search	BD	$[(\log_2 B) +$ #match pg]*D	$[(\log_F 1.5B) +$ #match pg]*D
Insert	2D	$((\log_2 B)+B)D$	$((\log_F 1.5B)+2)*D$
Delete	0.5BD + D	$((\log_2 B)+B)D$	$((\log_F 1.5B)+2)*D$



Cost of Operations

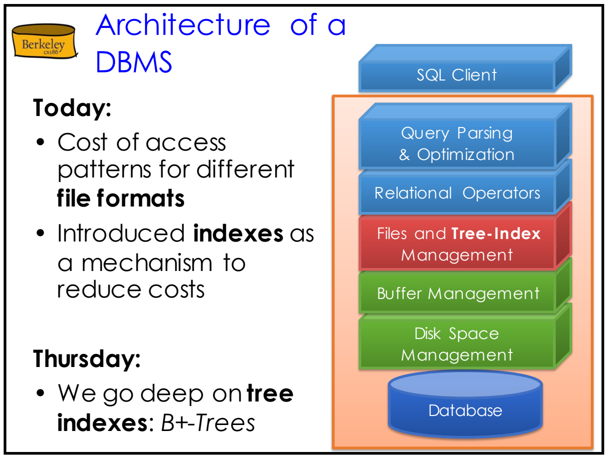
B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	Sorted File	Clustered Index
Scan all records	BD	BD	(3/2) BD = 1.5BD
Equality Search	0.5 BD	$(\log_2 B)$ * D	$(\log_F 1.5B+1)$ * D
Range Search	BD	[$(\log_2 B)$ + #match pg]*D	[$(\log_F 1.5B)$ + #match pg]*D
Insert	2D	$((\log_2 B)+B)$D	$((\log_F 1.5B)+2)$*D
Delete	0.5BD + D	$((\log_2 B)+B)$D	$((\log_F 1.5B)+2)$*D



Summary

- Many file orgs, with tradeoffs
  - Heap Files, Sorted Files, Clustered Files and Indexes
  - Benefits depend on the common operations
  - Compute expected costs
- Indexes: fast lookup of data entries by search key
  - Lookup: equivalence, range, region …
  - Search key: arbitrary columns
- Data entries:
  - 3 alternatives: By Value, By Reference, By List of References
- Often multiple indexes per file of data records
  - each with a different search key.
- Indexes can be classified as *clustered* vs. *unclustered*
  - Difs have important consequences for utility/performance.



Architecture of a DBMS

Today:

- Cost of access patterns for different **file formats**
- Introduced **indexes** as a mechanism to reduce costs

Thursday:

- We go deep on **tree indexes**: *B+-Trees*