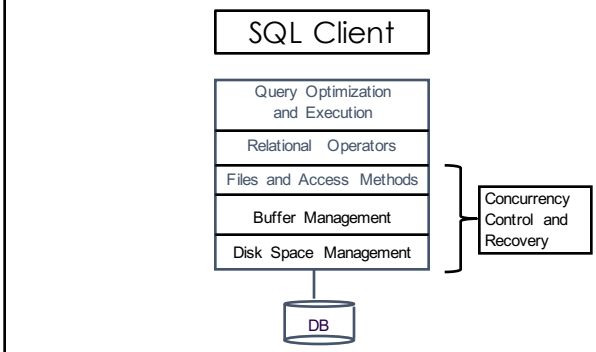
Storing Data: Disks, Buffers and Files

Berkeley cs186

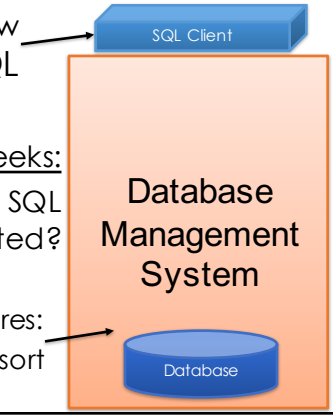
Block diagram of a DBMS



Last few lectures: SQL

Next few weeks: How is a SQL query Executed?

First few lectures: out-of-core sort

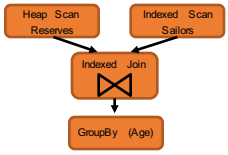


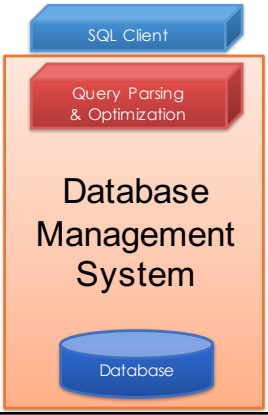
Architecture of a DBMS

Parse, check, and verify the SQL expression

```
SELECT s.sid, s.sname, r.bid
FROM Sailors s, Reserves r
WHERE s.sid = r.sid AND s.age > 30
```

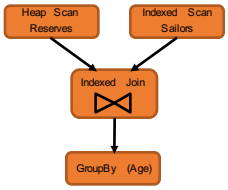
and translate into an efficient relational query plan

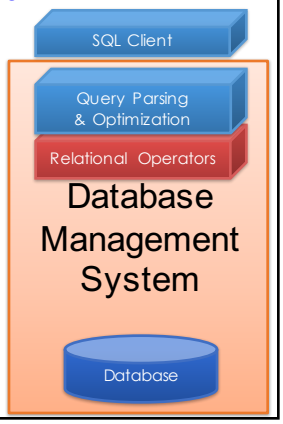




Architecture of a DBMS

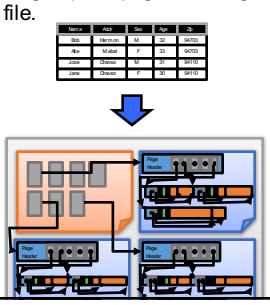
Execute the dataflow by operating on **records** and **files**

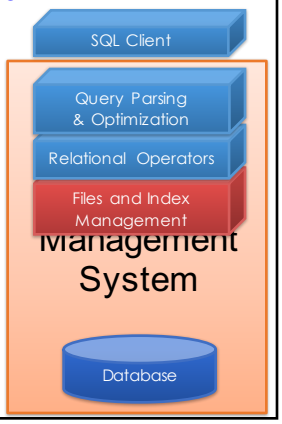




Architecture of a DBMS

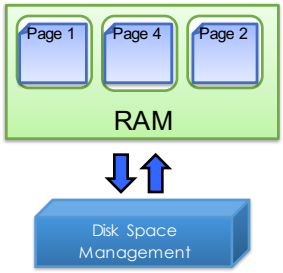
Organizing tables and records as groups of pages in a logical file.

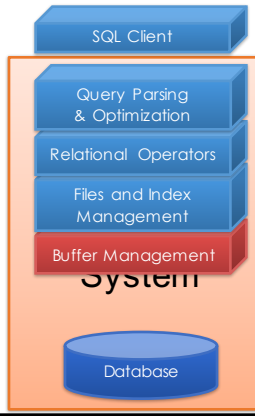




Architecture of a DBMS

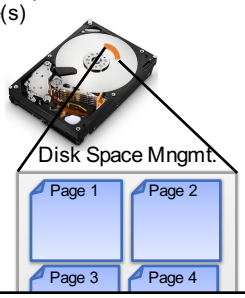
Illusion of operating in memory





Architecture of a DBMS

Translates page requests into physical bytes on one or more device(s)

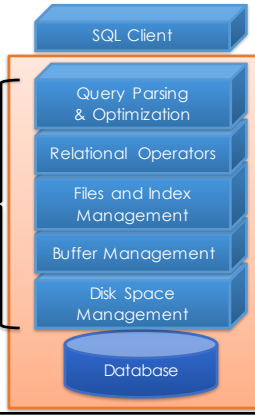


Architecture of a DBMS

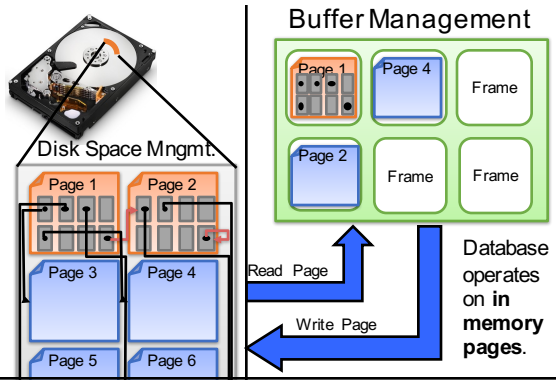
Organized in layers

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

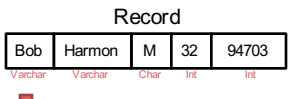
Example of **good** systems design

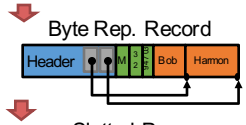


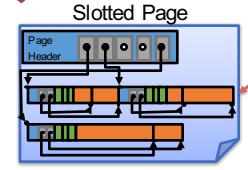
Disk Space & Buffer Management

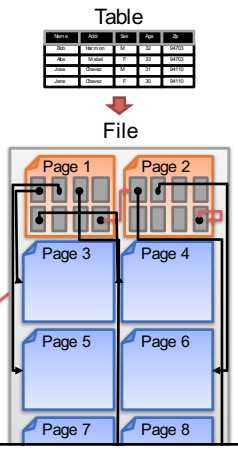


Overview







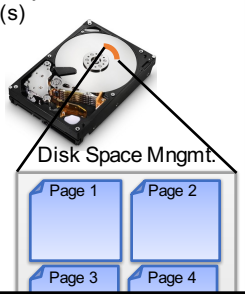


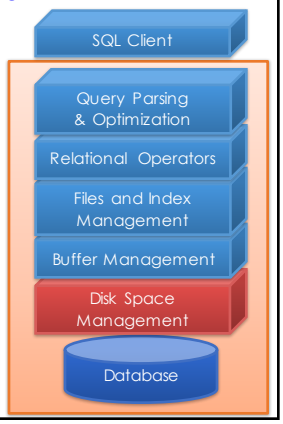
Overview: Files of Pages of Records

- Tables stored as a *logical files* consisting of *pages* each containing a collection of *records*
- Pages are managed
  - in memory by the buffer manager: higher levels of database only operate in memory
  - on disk by the disk space manager: reads and writes pages to physical disk/files
- Big Ideas in this Lecture
  - **Block/Pages** granularity reasoning
  - Exploit access patterns in **memory management**
  - Efficient **binary representation** of data

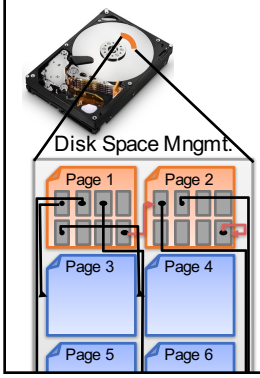
Architecture of a DBMS

Translates page requests into physical bytes on on or more device(s)



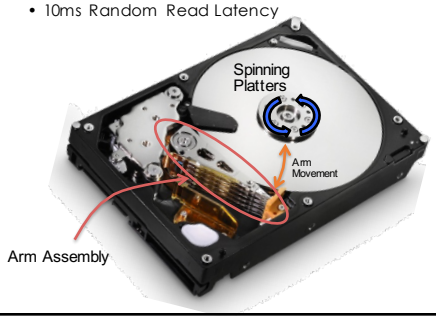


Disk Space Management



Recall: Disks and Files

- DBMS stores information on Disks and SSDs.
  - Disks are a mechanical anachronism (slow!)
    - 10ms Random Read Latency



Recall: Arranging Pages on Disk

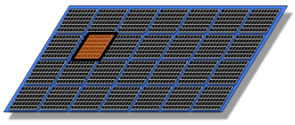
- "*Next*" page concept:
  - pages on same track, followed by
  - pages on same cylinder, followed by
  - pages on adjacent cylinder
- Arrange file pages sequentially on disk
  - minimize seek and rotational delay.
- For a sequential scan, *pre-fetch*
  - several pages at a time!
- Read large consecutive blocks

Recall: SSDs

- DBMS stores information on Disks and SSDs.
  - Disks are a mechanical anachronism (slow!)
  - SSDs faster, **slow relative to memory, costly writes**

Issues in current generation (NAND)
4-8K reads, **1-2MB writes**

So… read is fast and *predictable*
Single read access time: 0.03 ms
4KB random reads: ~500MB/sec
Sequential reads: ~525MB/sec



But.. write is not! Slower for random
Single write access time: 0.03ms
4KB random writes: ~120MB/sec
Sequential writes: ~480MB/sec
Write amplification: big units, need to reorg for garbage collection & wear

Disks and Files

- DBMS stores information on **Disks** and **SSDs**.
  - Disks are a mechanical **anachronism** (slow!)
  - SSDs faster, **slow relative to memory**, costly writes
- DBMS operate at **Block Level**
  - Read and Write large **chunks seq. bytes**
    - Leverage cache hierarchy and HW pre-fetch
    - Amortize seek delays on HDDs and Writes on SSD
  - *Sequentially*: *Next* disk block is fastest
  - Maximize usage of data per R/W
- Organize data for fast in memory processing (i.e., mapping)

A note on (confusing) terminology

Block = Unit of transfer for disk read/write
- 64KB - 128KB is a good number today
- Book says 4KB

Page = Fixed size contiguous chunk of memory
- Assume same size as block
- Refer to corresponding blocks on disk

For simplicity we use Block and Page interchangeably.

Disk Space Management

Lowest layer of DBMS, manages space on disk
- Mapping pages to locations on disk
- Loading pages from disk to memory
- Saving pages back to disk & ensuring writes

Higher levels call upon this layer to:
- read/write a pages
- allocate/de-allocate logical pages

Request for a *sequence* of pages best satisfied by pages stored sequentially on disk
- Physical details hidden from higher levels of system
- Higher levels may assume **Next Page** is fast!

Disk Space Management Implementation

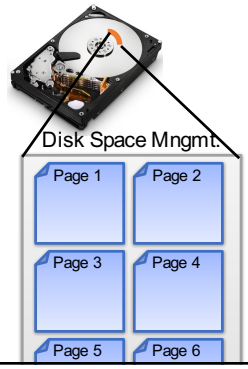
Proposal 1: Talk to the device directly
- Could be very fast if you knew the device well
- What happens when devices change?

Proposal 2: Run over filesystem (FS)
- Allocate single large "contiguous" file and assume sequential / nearby byte access are fast
- Most FS optimize for sequential access and temporal locality (buffer cache on hot items)
  - Sometimes disable FS buffering
- May span multiple files on multiple disks / machines

Typically sits on top of local file system

Get Page 4 Get Page 5

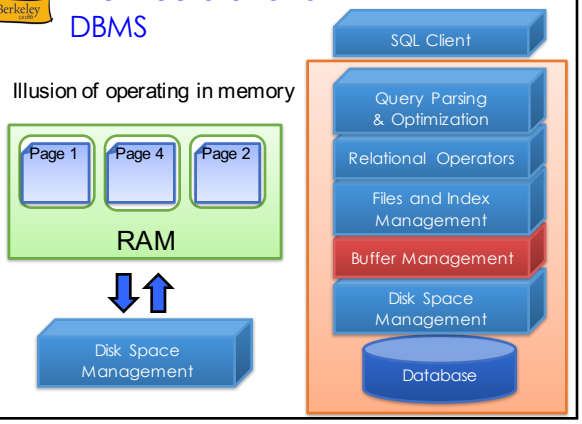
Disk Space Management



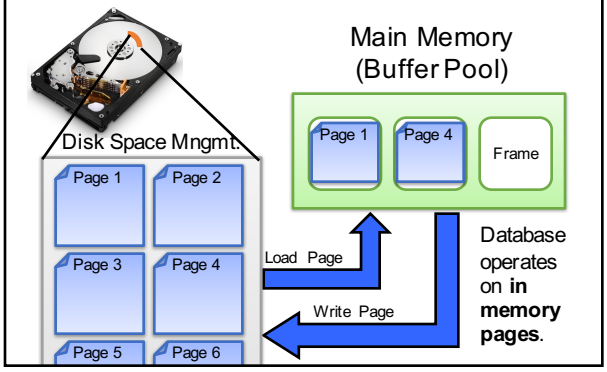
- Provide API to read and write pages to device
- Pages: block level organization of bytes on disk
- Ensures next locality and abstracts FS/Device details

Architecture of a DBMS

Illusion of operating in memory

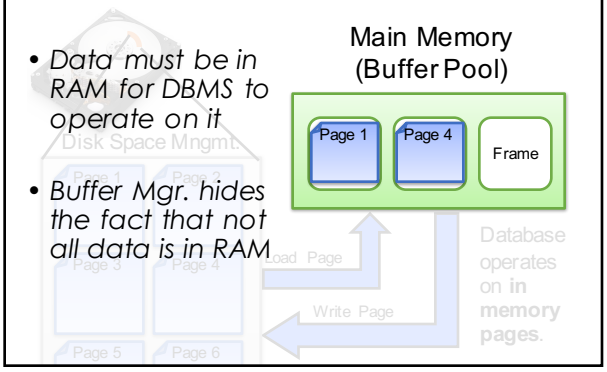


Mapping Pages into Memory

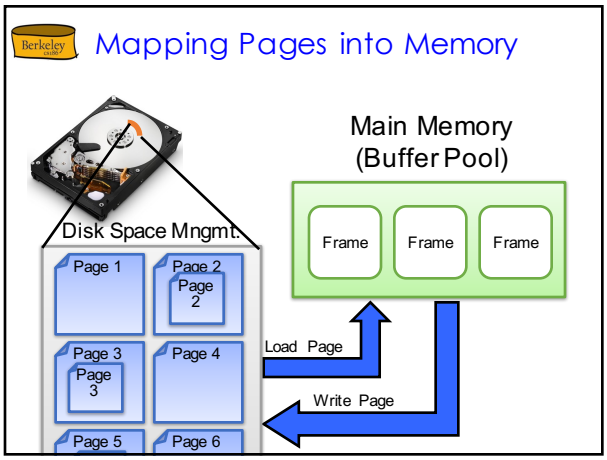


Mapping Pages into Memory

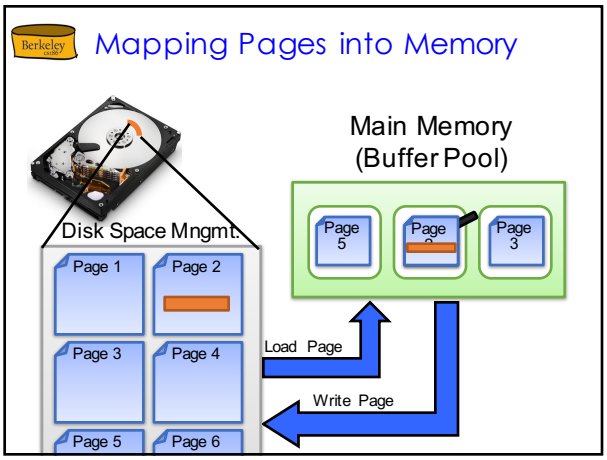
- *Data must be in RAM for DBMS to operate on it*
- *Buffer Mgr. hides the fact that not all data is in RAM*



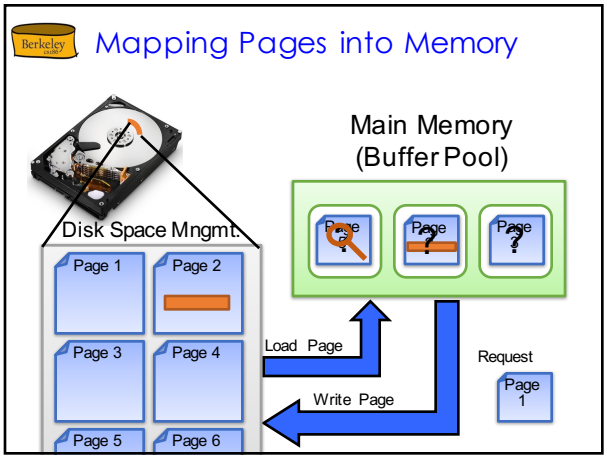
Mapping Pages into Memory



Mapping Pages into Memory



Mapping Pages into Memory



Challenges for Buffer Manager

- What happens if a page is modified?
  - Writing "dirty" pages back to disk
- No free frames → evict which page?
  - Replacement Policy
- What if page is being used?
  - Pinning
- Concurrent page operations?
  - Solve this in lock manager …

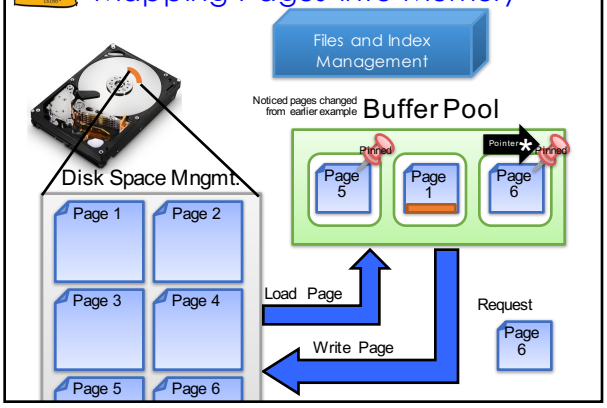
When a Page is Requested ...

- Buffer pool information "table" contains: <frame#, pageid, pin_count, dirty>

1. If requested page is not in pool:
   a. Choose a frame for *replacement*. *Only "un-pinned" pages are candidates!*
   b. If frame "dirty", write current page to disk
   c. Read requested page into frame
2. *Pin* the page and return its address.

If requests can be predicted (e.g., sequential scans) pages can be pre-fetched several pages at a time!

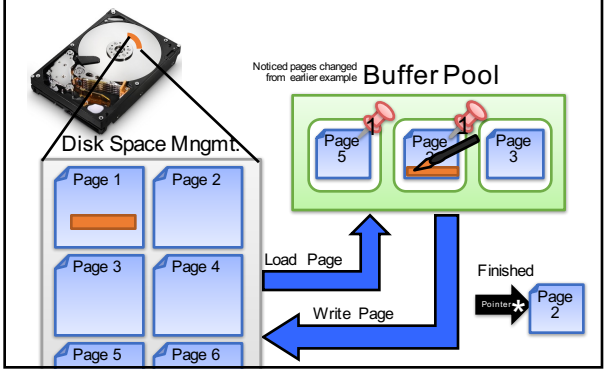
Mapping Pages into Memory



After Requestor Finishes

- Requestor of page must:
  1. indicate whether page was modified via *dirty* bit.
  2. *unpin* it (soon preferably!) why?
- Page in pool may be requested many times,
  - a *pin count* is used.
  - To pin a page: pin_count++
  - A page is a candidate for replacement iff *pin count* == 0 ("unpinned")
- CC & recovery may do additional I/Os upon replacement.
  - *Write-Ahead Log* protocol; more later!

Mapping Pages into Memory



Page Replacement Policy

- Page is chosen for replacement by a *replacement policy:*
  - Least-recently-used (LRU), Clock
  - Most-recently-used (MRU)
- Policy can have big impact on #I/O's;
  - Depends on the *access pattern.*

LRU Replacement Policy

- *Least Recently Used (LRU)*
  - Pinned Frame: not available to replace
  - track time each frame last *unpinned* (end of use)
  - replace the frame which least recently unpinned
- Very common policy: intuitive and simple
  - Works well for repeated accesses to popular pages (temporal locality)
  - *Can be costly. Why?*
    - *Need to maintain heap data-structure*
    - *Solution?*

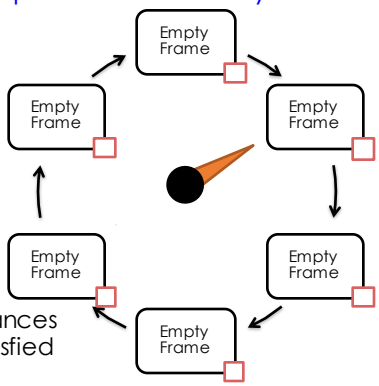
Clock Replacement Policy

Approx. to LRU

Arrange frames in logical cycle

Introduce "Ref. Bit"

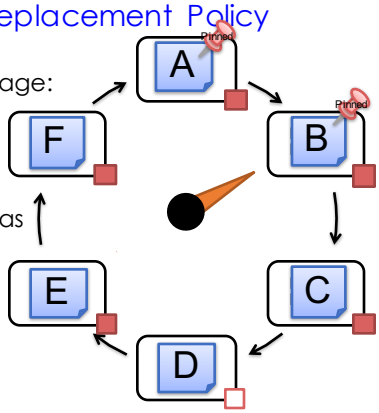
Clock arm advances until request satisfied



Clock Replacement Policy

Want to insert page: G

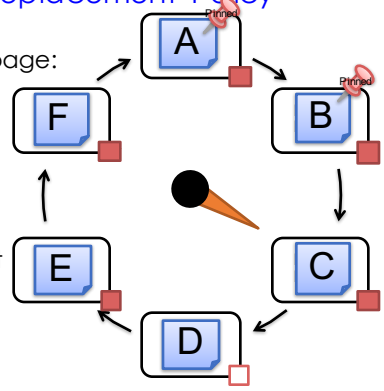
Current frame has pin-count > 0
→ Skip



Clock Replacement Policy

Want to insert page: G

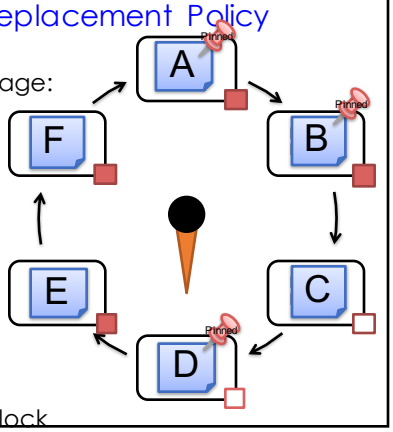
Not Pinned and Ref. bit is set →
1. Clear Ref. Bit
2. Skip



Clock Replacement Policy

Want to insert page: G

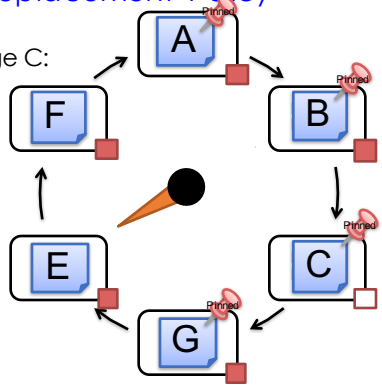
Not pinned and Ref. bit unset:
1. Evict
2. Copy Page
3. Set pinned
4. Set ref. bit
5. Advanced clock



Clock Replacement Policy

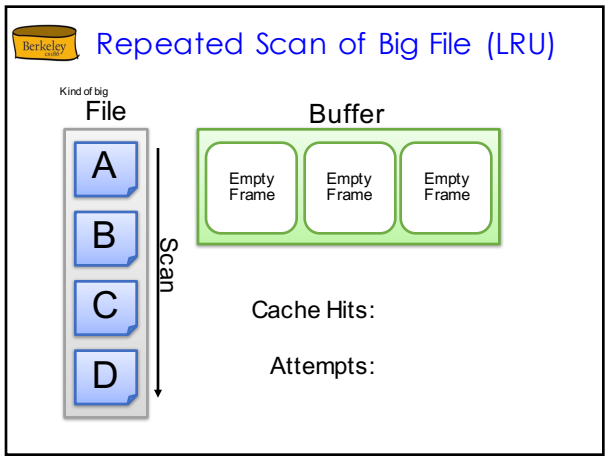
Request for Page C:

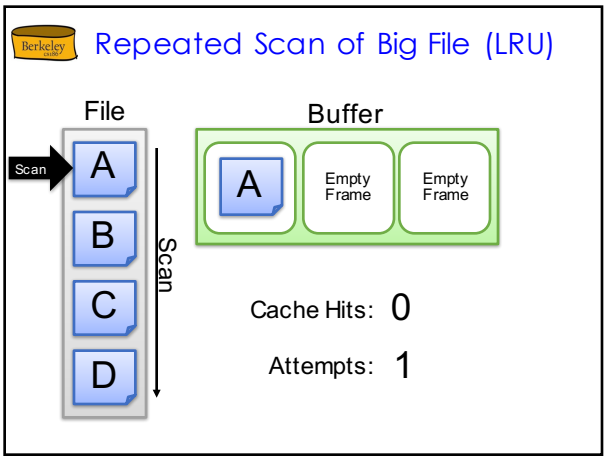
Cache Hit!:
1. Pin (inc.)
2. Set ref bit.

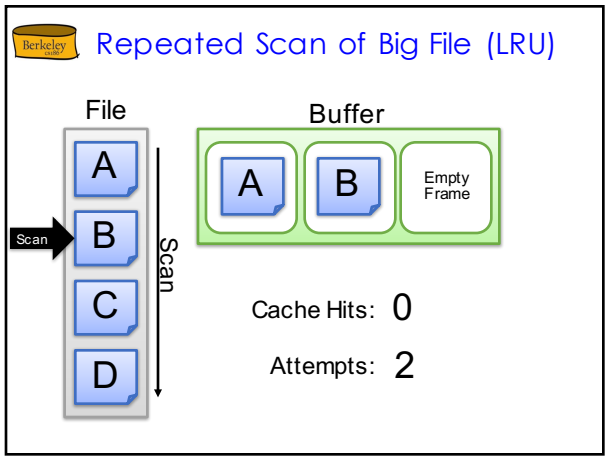


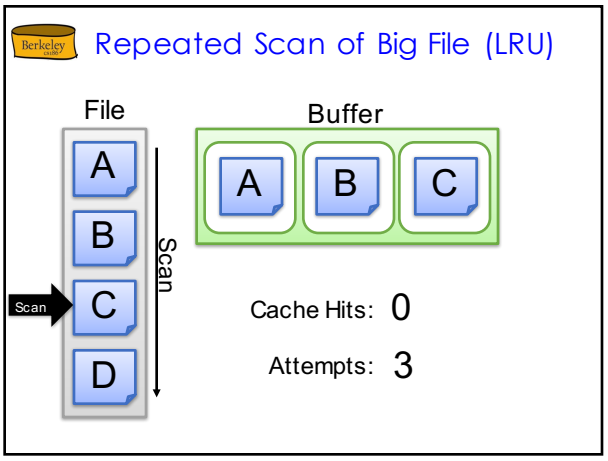
Is LRU/Clock Always Best?

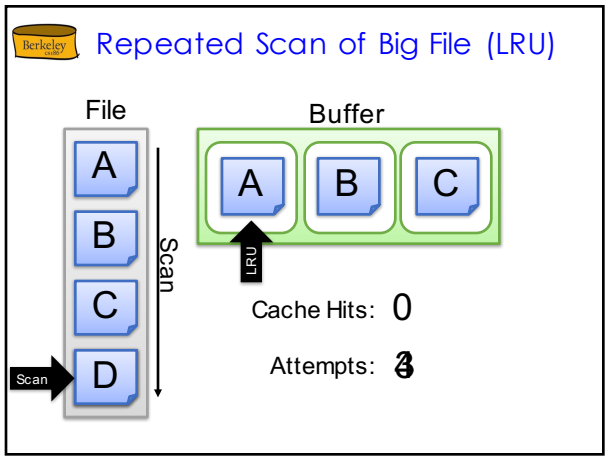
- Very common policy: *intuitive* and *simple*
- Works well for repeated accesses to popular pages
  - temporal locality
- LRU can be costly → Clock policy is cheap
- When might it perform poorly
  - *What about repeated scans of big files?*

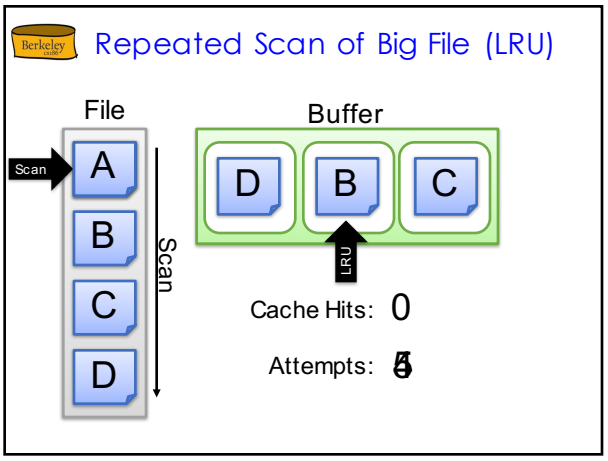

Repeated Scan of Big File (LRU)
Kind of big
File
Buffer
A
B
C
D
Scan
Empty Frame
Empty Frame
Empty Frame
Cache Hits:
Attempts:

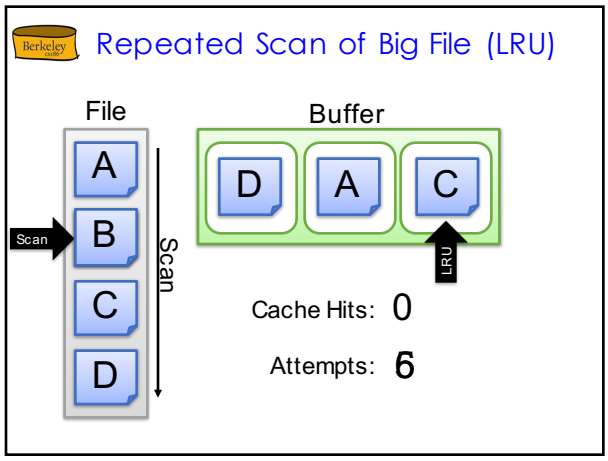

Repeated Scan of Big File (LRU)
File
Buffer
Scan
A
B
C
D
A
Empty Frame
Empty Frame
Cache Hits: 0
Attempts: 1

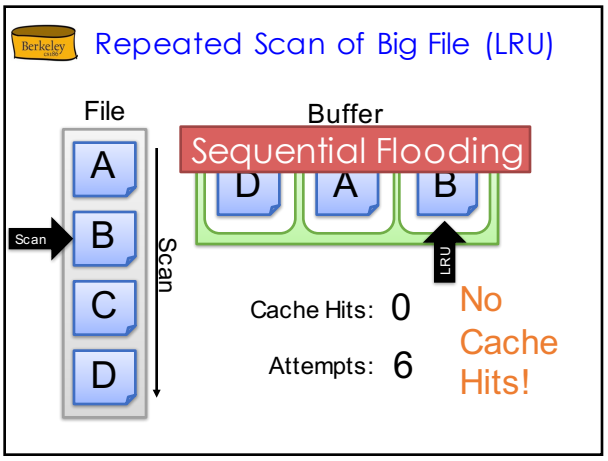

Repeated Scan of Big File (LRU)
File
Buffer
Scan
A
B
C
D
A
B
Empty Frame
Cache Hits: 0
Attempts: 2

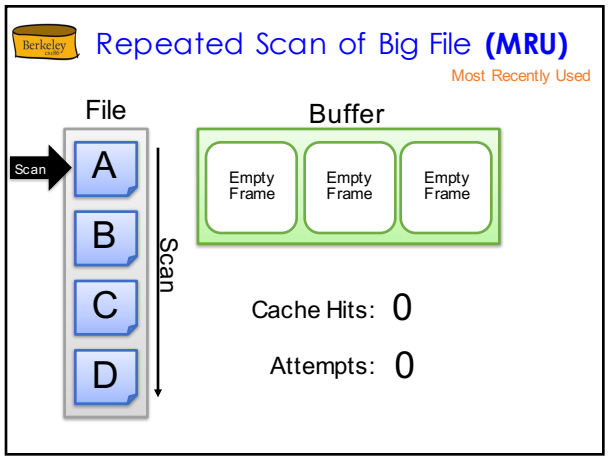

Repeated Scan of Big File (LRU)
File
Buffer
Scan
A
B
C
D
A
B
C
Cache Hits: 0
Attempts: 3

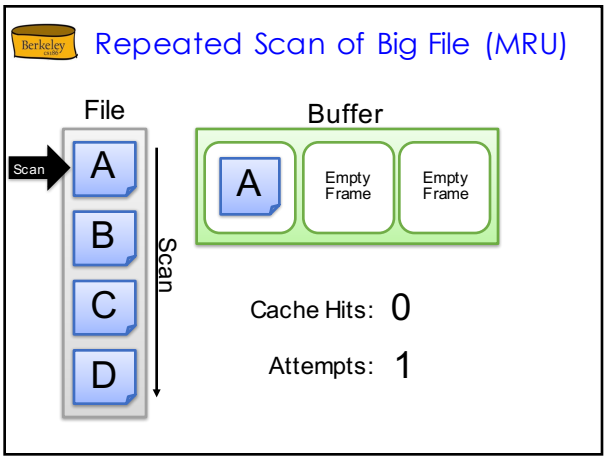

Repeated Scan of Big File (LRU)
File
Buffer
Scan
A
B
C
D
A
B
C
LRU
Cache Hits: 0
Attempts: 4

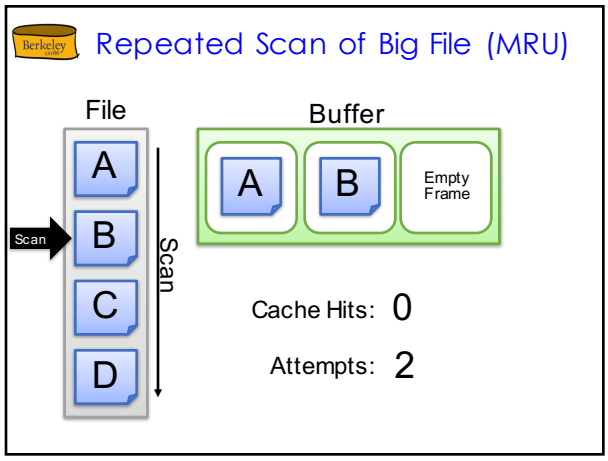

Repeated Scan of Big File (LRU)
File
Buffer
Scan
A
B
C
D
D
B
C
LRU
Cache Hits: 0
Attempts: 5

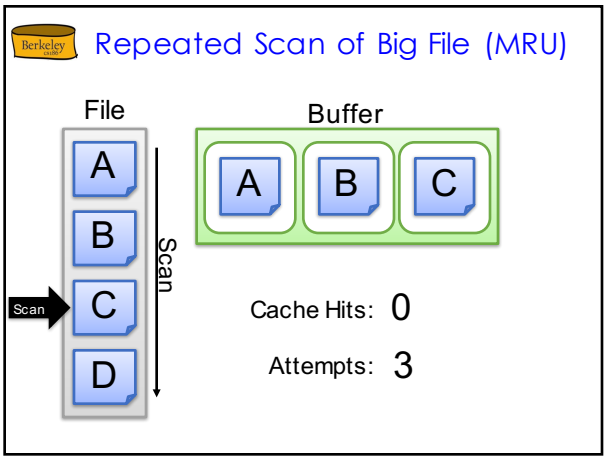

Repeated Scan of Big File (LRU)
File
Buffer
A
B
C
D
Scan
D
A
C
LRU
Cache Hits: 0
Attempts: 6

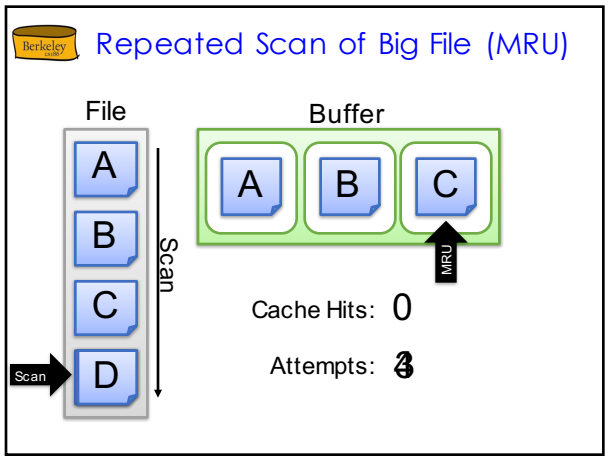

Repeated Scan of Big File (LRU)
File
Buffer
Sequential Flooding
A
B
C
D
Scan
D
A
B
LRU
Cache Hits: 0
Attempts: 6
No Cache Hits!

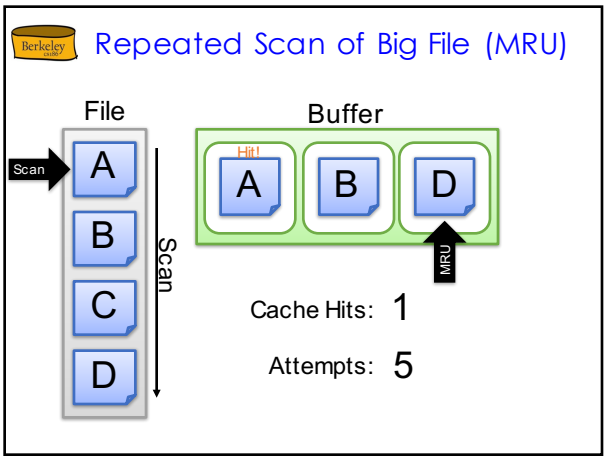

Repeated Scan of Big File (MRU)
Most Recently Used
File
Buffer
A
B
C
D
Scan
Empty Frame
Empty Frame
Empty Frame
Cache Hits: 0
Attempts: 0

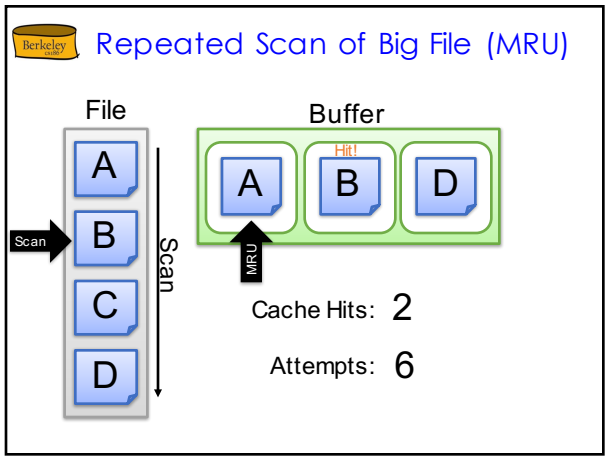

Repeated Scan of Big File (MRU)
File
Buffer
A
B
C
D
Scan
A
Empty Frame
Empty Frame
Cache Hits: 0
Attempts: 1

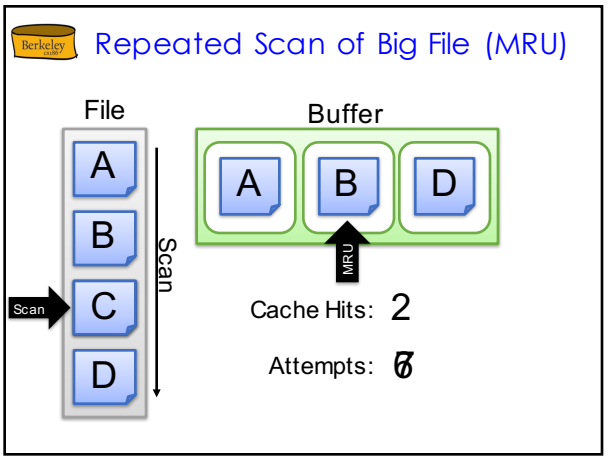

Repeated Scan of Big File (MRU)
File
Buffer
A
B
C
D
Scan
A
B
Empty Frame
Cache Hits: 0
Attempts: 2

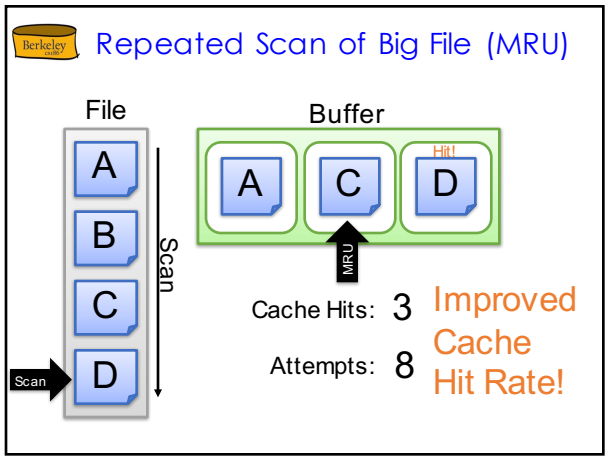

Repeated Scan of Big File (MRU)
File
Buffer
A
B
C
D
Scan
A
B
C
Cache Hits: 0
Attempts: 3

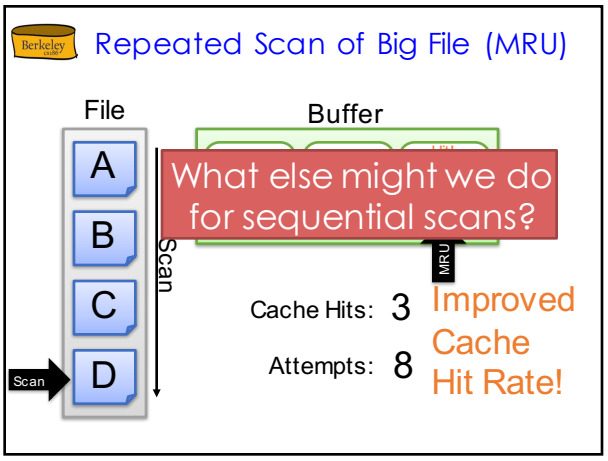

Repeated Scan of Big File (MRU)
File
Buffer
A
B
C
D
Scan
MRU
Cache Hits: 0
Attempts: 3


Repeated Scan of Big File (MRU)
File
Buffer
Hit!
Scan
Cache Hits: 1
Attempts: 5

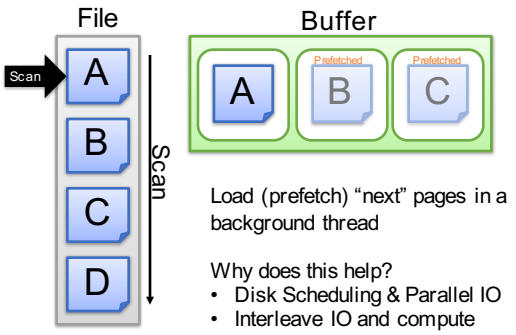

Repeated Scan of Big File (MRU)
File
Buffer
Hit!
Scan
Cache Hits: 2
Attempts: 6


Repeated Scan of Big File (MRU)
File
Buffer
Scan
Cache Hits: 2
Attempts: 7


Repeated Scan of Big File (MRU)
File
Buffer
Hit!
Scan
Cache Hits: 3
Attempts: 8
Improved Cache Hit Rate!


Repeated Scan of Big File (MRU)
File
Buffer
What else might we do for sequential scans?
Scan
Cache Hits: 3
Attempts: 8
Improved Cache Hit Rate!

Background Prefetching



Load (prefetch) "next" pages in a background thread

Why does this help?
- Disk Scheduling & Parallel IO
- Interleave IO and compute

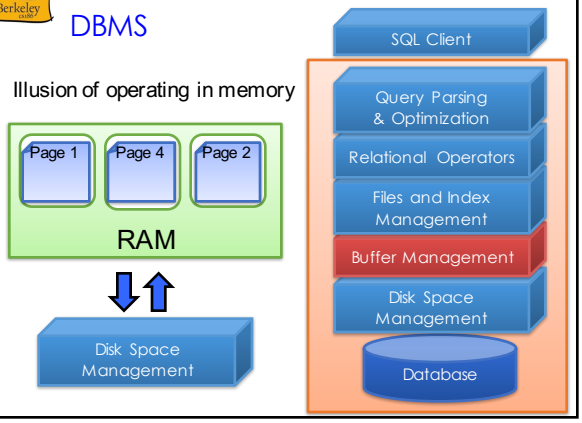
DBMS vs. OS Buffer Cache

OS has page/buffer management. Why not let OS manage these tasks?

Buffer management in DBMS requires ability to:
- pin page in buffer pool, force page to disk, order writes
  - important for implementing CC & recovery
- adjust *replacement policy*, and pre-fetch pages based on access patterns in typical DB operations.

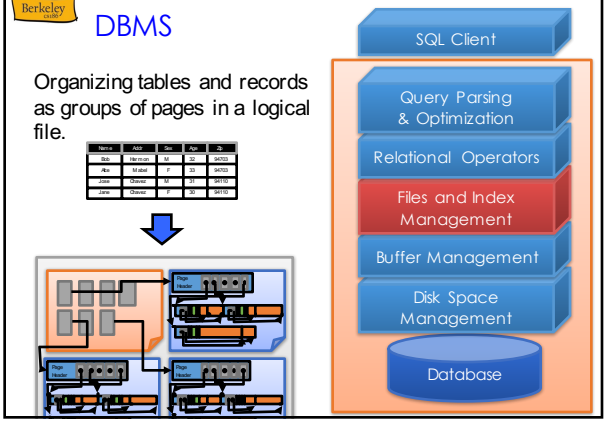
Architecture of a DBMS

Illusion of operating in memory

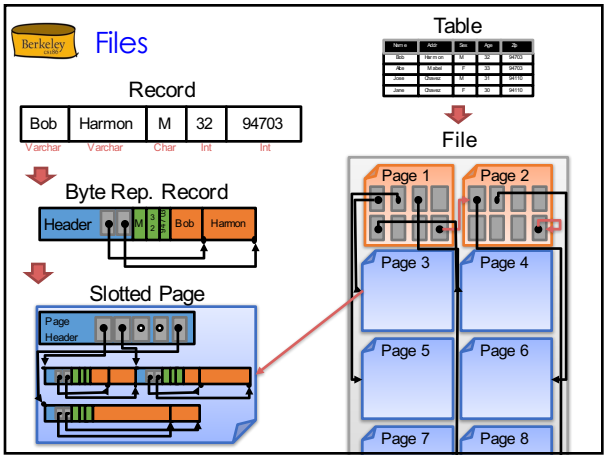


Architecture of a DBMS

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



Files



Files of Pages of Records

- Higher levels of DBMS operate on *pages of records* and *files of pages*.
- FILE: A collection of pages, each containing a collection of records. Must support:
  - insert/delete/modify record
  - fetch a particular record by ***record id*** …
    - *Think: pointer encoding Page_ID and location on page.*
  - scan all records (possibly with some conditions on the records to be retrieved)
- Could span multiple OS files and even machines
  - Or "raw" disk devices

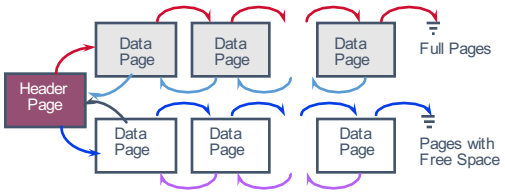
Many Kinds of Database Files

- Unordered Heap Files
  - Records placed arbitrarily across pages
- Clustered Heap File and Hash Files
  - Records and pages are grouped
- Sorted Files
  - Page and records are in sorted order
- Index Files
  - B+ Trees, Hash Tables, …
  - May contain records or point to records in other files

Basic Unordered Heap Files

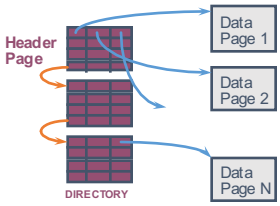
- Collection of records in no particular order
  - Not to be confused with "heap data-structure"
- As file shrinks/grows, pages (de)allocated
- To support record level operations, we must:
  - keep track of the *pages* in a file
  - keep track of *free space* on pages
  - keep track of the *records* on a page
- There are many alternatives for keeping track of this, we'll consider two in this class

Heap File Implemented as a List



- Header page ID and Heap file name stored elsewhere
  - Database "catalog"
- Each page contains 2 "pointers" plus **free-space** and **data**.
- What is wrong with this?
  - How do I find a page with enough space for a 20 byte record?

Better: Use a Page Directory



- Directory entries include **#free bytes** on the page.
- Header pages accessed often → likely in cache
  - What eviction policy is best here?
- Finding a page to fit a record required far fewer page loads than linked list (Why?)
  - One header page load reveals free space of many pages.

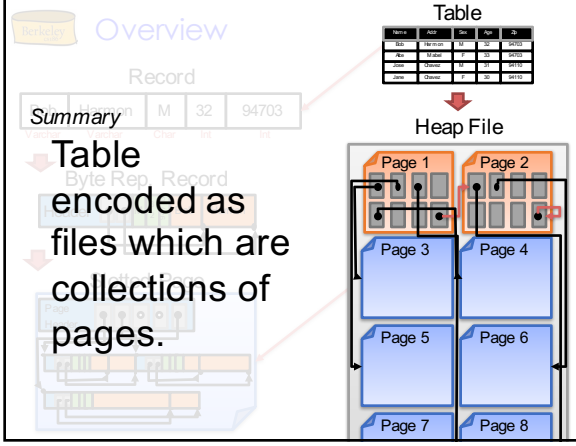
Indexes (sneak preview)

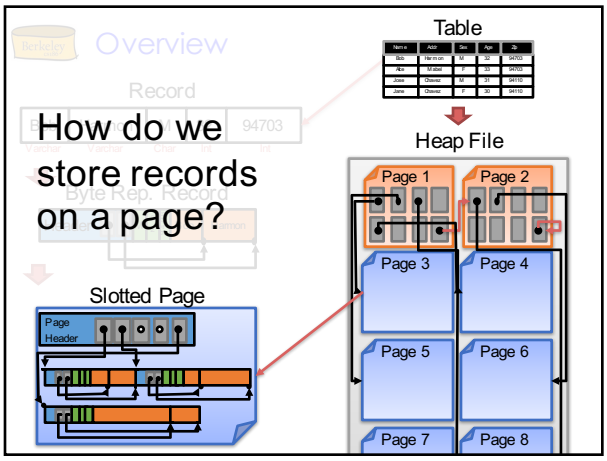
- A Heap file allows us to retrieve records:
  - by specifying the *record id (page id + slot)*
  - by scanning all records sequentially
- Would like to fetch records *by value*, e.g.,
  - Find all students in the "CS" department
  - Find all students with a gpa > 3 AND blue hair
- Indexes: file structures for efficient value-based queries

Overview

Summary

Table encoded as files which are collections of pages.





Page Basics: The Header

Blank Page
128KB

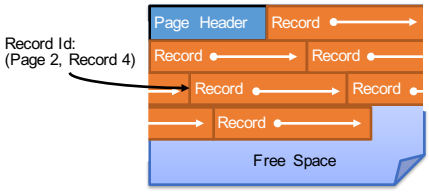
Page Basics: The Header

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

Things to Consider

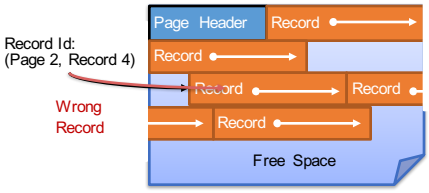
- Record length? *Fixed* or *Variable*
- Find records by record id? *Offset…*
- How do we add and delete records?
  - Bitmaps & Slot Tables

Fixed Length Records: Packed



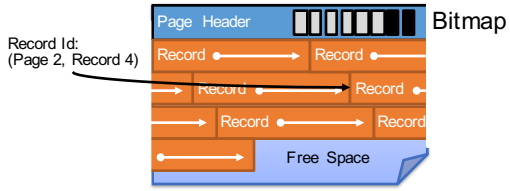
- Pack records densely
- Record id: record number in page
- Easy to add: just append
- Delete?

Fixed Length Records: Packed



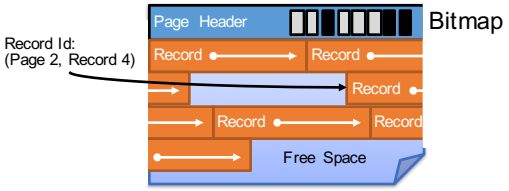
- Pack records densely
- Record id: record number in page
- Easy to add: just append
- Delete? Re-arrange …

Fixed Length Records: Unpacked



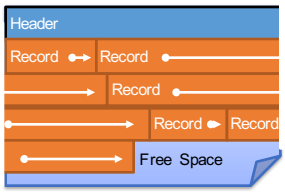
- Bitmap denotes "slots" with records
- Record id: record number in page
- **Insert:** find first empty slot
- **Delete?**

Fixed Length Records: Unpacked



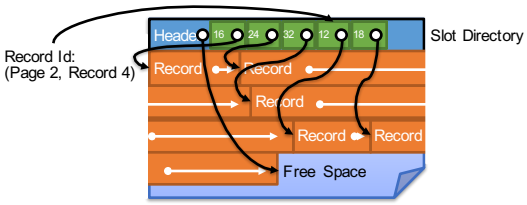
- Bitmap denotes "slots" with records
- Record id: record number in page
- **Insert:** find first empty slot
- **Delete:** Clear bit

Variable Length Records



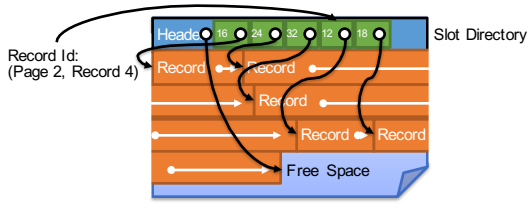
- How do we know where each record begins?
- What happens when we add and delete records?

Slotted Page



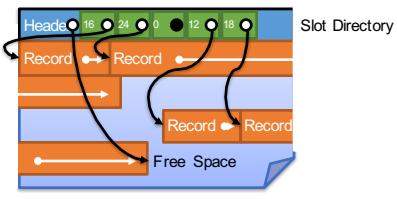
- Introduce **slot directory** in header
  - Length + Pointer to beginning of record
  - Pointer to free space
- Record ID = location in slot table
- ***Delete***? (e.g., 3rd record on the page)

Slotted Page



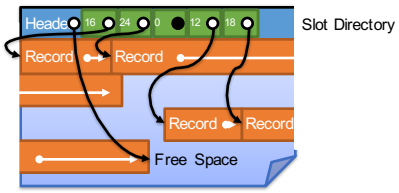
- **Delete**: Set pointer to null.
  - Doesn't affect pointers to other records
  - However, need to make sure we remove any references to **record_id** in indexes

Slotted Page



- ***Insert***? 

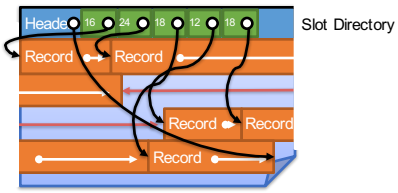
Slotted Page



- **Insert**:
  - Place record in free space on page

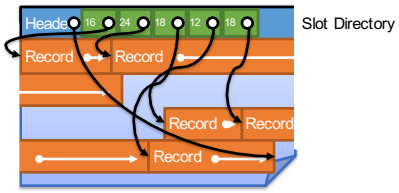
Record

Slotted Page



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

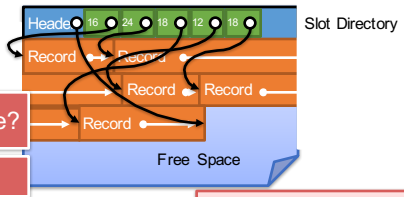
Slotted Page



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

Is this safe?

Slotted Page



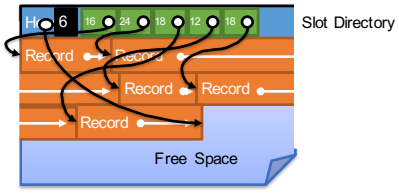
Is this safe?

When?

What if we need more slots?

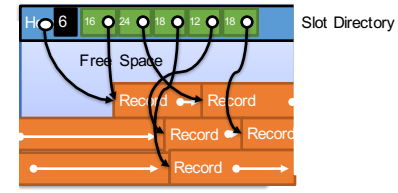
- **Insert**:
  - Place record in free s
  - Create pointer in next open slot in slot directory
  - Reorganize data on page.

Slotted Page: Growing Slots



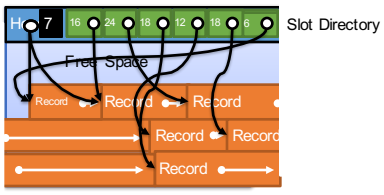
- Track number of slots in slot directory

Slotted Page: Growing Slots



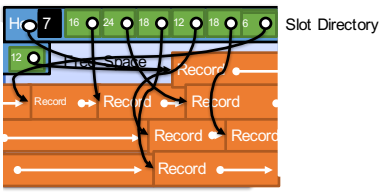
- Track number of slots in slot directory
- Grow records from other end of page
  - Why?

Slotted Page: Growing Slots



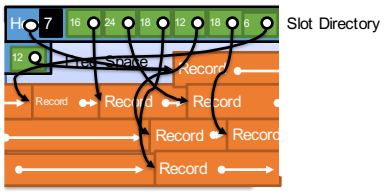
- Track number of slots in slot directory
- Grow records from other end of page
- Extend slot directory on insert
  - Add record in free space & update counter

Slotted Page: Growing Slots



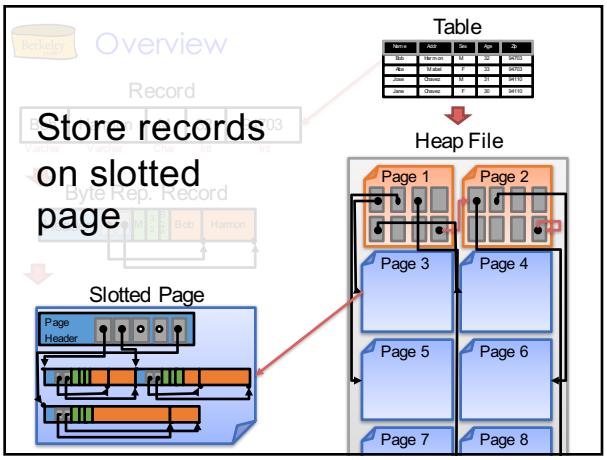
- Track number of slots in slot directory
- Grow records from other end of page
- Extend slot directory on insert
  - Add record in free space & update counter

Slotted Page Summary

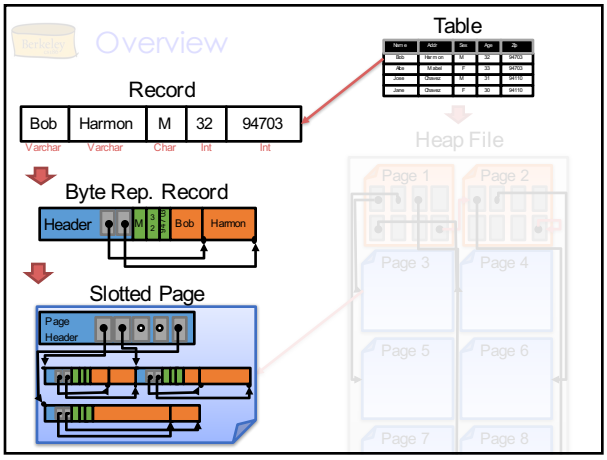


- Typically use slotted Page
  - Good for variable and fixed length records
- Good for fixed length records too. Why?
  - Re-arrange (e.g., sort) and squash null fields

Overview



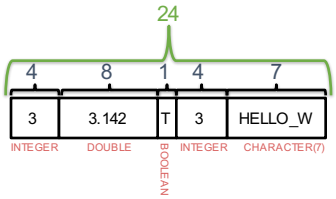
Overview



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*

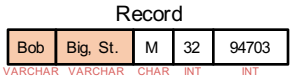
Record Formats: Fixed Length



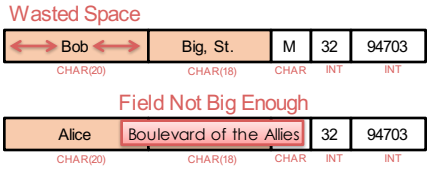
- Field types same for all records in a file.
  - Type info stored separately in *system catalog*
- On disk byte representation same as in memory
- Finding *i'th* field?
  - done via arithmetic (fast)
- Compact? (Nulls?)

Record Formats: Variable Length

What happens if fields are variable length?

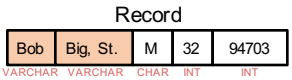


Could store with padding? (Fixed Length)

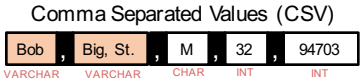


Record Formats: Variable Length

What happens if fields are variable length?



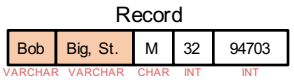
Could use delimiters (i.e., CSV):



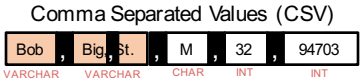
- Issues?

Record Formats: Variable Length

What happens if fields are variable length?



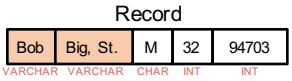
Could use delimiters (i.e., CSV):



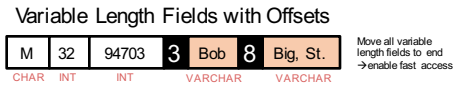
- Requires scan to access field
- What if text contains commas?

Record Formats: Variable Length

What happens if fields are variable length?



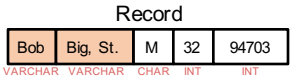
Store length information before fields:



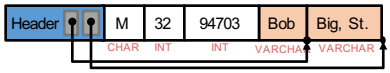
- Requires scan to access field
- ~~What if text contains commas?~~

Record Formats: Variable Length

What happens if fields are variable length?



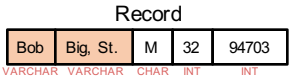
Introduce a record header:



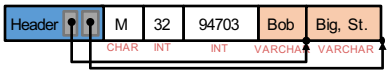
- ~~Requires scan to access field~~. Why?
- ~~What if text contains commas?~~

Record Formats: Variable Length

What happens if fields are variable length?

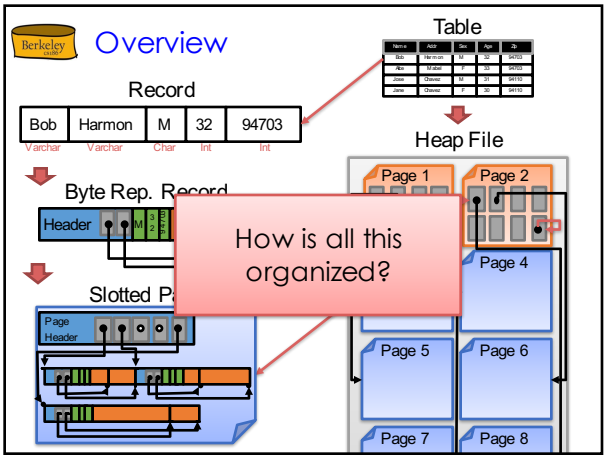


Introduce a record header:



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

Overview



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

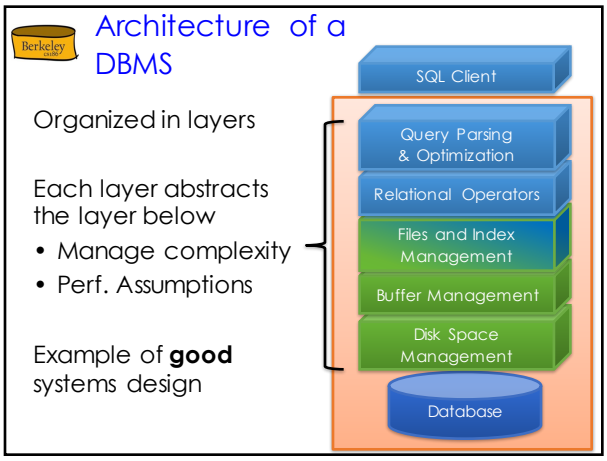
```
postgres=# \d pg_attribute
    Table "pg_catalog.pg_attribute"
    Column     |   Type    | Modifiers
---------------+-----------+-----------
 attrelid      | oid       | not null
 attname       | name      | not null
 atttypid      | oid       | not null
 attstattarget | integer   | not null
 attlen        | 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:~$
```

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
SQL Client
Query Parsing & Optimization
Relational Operators
Files and Index Management
Buffer Management
Disk Space Management
Database