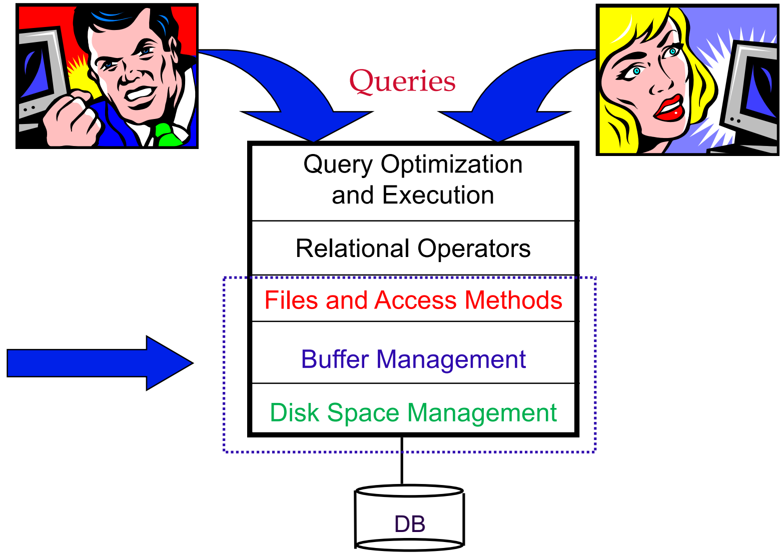
Database Management Systems

Buffer and File Management

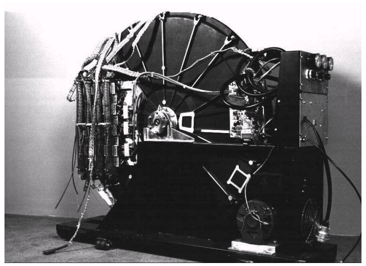
Fall 2015

"Yea, from the table of my memory
I'll wipe away all trivial fond records."
-- Shakespeare, *Hamlet*

The BIG Picture



Disks and Files



- DBMS stores information on disks.
  - In an electronic world, disks are a mechanical anachronism!
- This has major implications for DBMS design!
  - READ: transfer data from disk to main memory (RAM).
  - WRITE: transfer data from RAM to disk.
  - Both are high-cost operations, relative to in-memory operations, so must be planned carefully!

Why Not Store It All in Main Memory?

- *Costs too much*. $60 will buy you either around 8 GB of RAM or around 1000 GB (1 TB) of disk today.
  - High-end Databases today can be in the Petabyte (1000TB) range.
  - Approx 60% of the cost of a production system is in the disks.
- *Main memory is volatile*. We want data to be saved between runs. (Obviously!)
- Note, some specialized systems do store entire database in main memory.
  - Vendors claim 10x speed up vs. traditional DBMS running in main memory.

Storage Access

- A database file is partitioned into fixed-length storage units called **blocks**
  - Units of both storage allocation and data transfer.
  - Database system seeks to minimize the number of block transfers between the disk and memory.
  - Can reduce the number of disk accesses by keeping as many blocks as possible in main memory.
- **Buffer** – portion of main memory available to store copies of disk blocks.
- **Buffer manager** – subsystem responsible for allocating buffer space in main memory.

More Terminology…

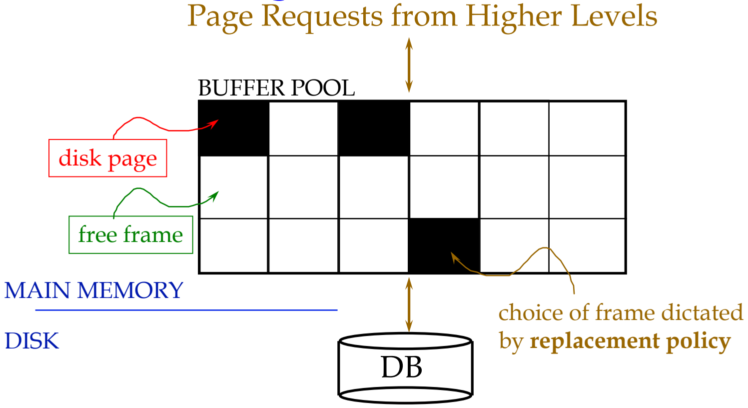
- Disk Page – the unit of transfer between the disk and memory

  Typically set as a config parameter for the DBMS.
  Typical value between 4 KBytes to 32 KBytes.
- Frame – a unit of memory

  Typically the same size as the Disk Page Size
- Buffer Pool –
  - *An area of memory into which database pages are read, modified, and held during processing*
  - A collection of frames used by the DBMS to temporarily keep data for use by the query processor.
    - note: We will sometime use the term "buffer" and "frame" synonymously.
- Pinned block – memory block that is not allowed to be written back to disk.

Question: When would you use a larger page size rather than a smaller one?

Buffer Management in a DBMS



- *Data must be in RAM for DBMS to operate on it!*
  - *The query processor refers to data using virtual memory addresses.*
- *Buffer Mgr hides the fact that not all data is in RAM*

When a Page is Requested ...

- If requested page IS in the pool:
  - *Pin* the page and return its address.
- Else, if requested page IS NOT in the pool:
  - If a free frame exists, choose it, Else:
    - Choose a frame for *replacement (only un-pinned pages are candidates)*
    - If chosen frame is "dirty", write it to disk
  - Read requested page into chosen frame
  - *Pin* the page and return its address.

Buffer Control Blocks (BCBs):

<frame#, pageid, pin_count, dirty>

- A page may be requested many times, so
  - a *pin count* is used.
  - To pin a page, pin_count++
  - A page is a candidate for replacement iff *pin_count* == 0 (*"unpinned"*)
- Requestor of page must eventually unpin it.
  - pin_count--
- Must also indicate if page has been modified:
  - *dirty* bit is used for this.

  Q: Why is this important?

Additional Buffer Manager Notes

- BCB's are hash indexed by pageID
- Concurrency Control & Recovery may entail additional I/O when a frame is chosen for replacement.

  (*Write-Ahead Log* protocol; more later.)
- If requests can be predicted (e.g., sequential scans) pages can be pre-fetched several pages at a time.

Buffer Replacement Policy

- Frame is chosen for replacement by a *replacement policy:*
  - Least-recently-used (LRU), MRU, Clock, etc.
- This policy can have big impact on the number of disk reads and writes.
  - Remember, these are slooooooooooow.
- **BIG IDEA** – throw out the page that you are least likely to need in the future.
  - Q: How do you predict the future?
- Efficacy depends on the *access pattern.*

LRU Replacement Policy

- *Least Recently Used (LRU)*

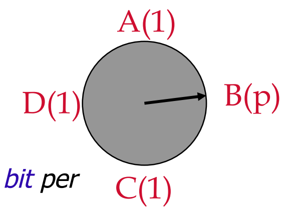
1) for each page in buffer pool, keep track of time last *unpinned*
   - *What else might you keep track off?*
   - *How would that impact performance?*

2)Replace the frame that has the oldest (earliest) time
   - Most common policy: intuitive and simple
     - Based on notion of "Temporal Locality"
     - Works well to keep "working set" in buffers.
   - Implemented through doubly linked list of BCBs
     - Requires list manipulation on unpin

Some issues with LRU

- *Problem: Sequential flooding*
  - LRU + repeated sequential scans.
  - # buffer frames < # pages in file means each page request causes an I/O. *MRU* much better in this situation (but not in all situations, of course).
- *Problem: "cold" pages can hang around a long time before they are replaced*
  - Cold pages are pages that have been touched only once recently

"Clock" Replacement Policy



- An approximation of LRU
- Arrange frames into a cycle, store one *reference bit per frame*
  - Can think of this as the 2nd chance bit
- When pin count reduces to 0, turn on ref. bit
- When replacement necessary

```
do for each page in cycle {
        if (pincount == 0 && ref bit is on)
                turn off ref bit;
        else if (pincount == 0 && ref bit is off)
                choose this page for
                        replacement;
} until a page is chosen;
```

Questions:
How like LRU?
Problems?

"2Q" Replacement Policy

- One Queue (A1) has pages that have been referenced only once.
  - new pages enter here
- A second, LRU Queue (Am) has pages that have been referenced (pinned) multiple times.
  - pages get promoted from A1 to here
- Replacement victims are usually taken from A1
  - Q: Why????

DBMS vs. OS File System

OS does disk space & buffer mgmt: why not let OS manage these tasks?

- Some limitations, e.g., files can't span disks.
  - Note, this is changing --- OS File systems are getting smarter (i.e., more like databases!)
- Buffer management in DBMS requires ability to:
  - pin a page in buffer pool, force a 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.
- Q: Compare DBMS Buffer Mgmt to OS Virtual Memory? to Processor Cache?

Files of Records

- Blocks interface for I/O, but…
- Higher levels of DBMS operate on *records*, and *files of records*.
- FILE: A collection of pages, each containing a collection of records. Must support:
  - **insert**/**delete**/**modify** record
  - **fetch** a particular record (specified using *record id*)
  - **scan** all records (possibly with some conditions on the records to be retrieved)
- Note: typically
  page size = block size = frame size.

"MetaData" - System Catalogs

- How to impose structure on all those bytes??
- MetaData: "Data about Data"
- 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.
  - ☛ *Q: But how to store the catalogs????*

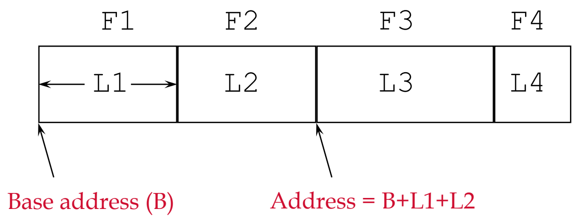
Catalogs are Stored as Relations!

attr_name	rel_name	type	position
attr_name	Attribute_Cat	string	1
rel_name	Attribute_Cat	string	2
type	Attribute_Cat	string	3
position	Attribute_Cat	integer	4
sid	Students	string	1
name	Students	string	2
login	Students	string	3
age	Students	integer	4
gpa	Students	real	5
fid	Faculty	string	1
fname	Faculty	string	2
sal	Faculty	real	3

Attr_Cat(attr_name, rel_name, type, position)

It's a bit more complicated…

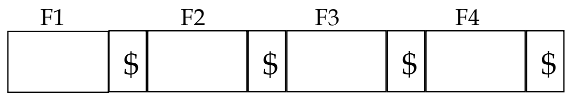
Record Formats: Fixed Length



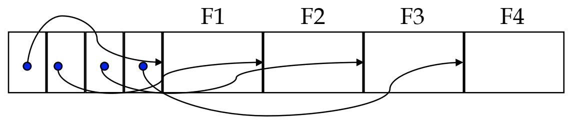
- Information about field types same for all records in a file; stored in *system catalogs.*
- Finding *i'th* field done via arithmetic.

Record Formats: Variable Length

- Two alternative formats (# fields is fixed):



Fields Delimited by Special Symbols



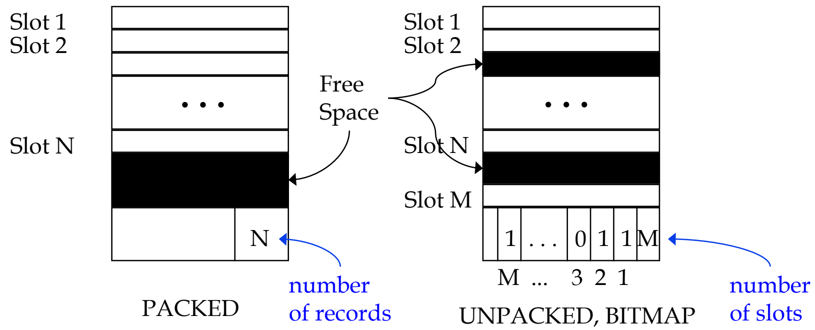
Array of Field Offsets

☛ Second offers direct access to i'th field, efficient storage of *nulls* (special *don't know* value); small directory overhead.

How to Identify a Record?

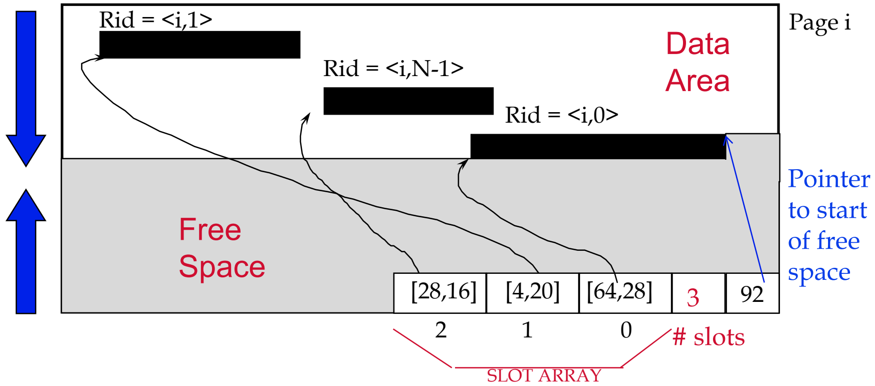
- The Relational Model doesn't expose "pointers", but that doesn't mean that the DBMS doesn't use them internally.
- Q: Can we use memory addresses to "point" to records?
- Systems use a "Record ID" or "RecID"

Page Formats: Fixed Length Records



☛ *Record id = <page id, slot #>. In first alternative, moving records for free space management changes rid; may not be acceptable.*

“Slotted Page” for Variable Length Records

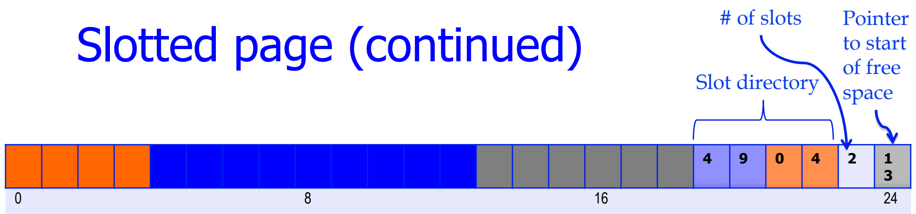


- Slot contains: [offset (from start of page), length]
  - in bytes
- *Record id = <page id, slot #>*
- Page is full when data space and slot array meet.

Slotted Page (continued)

- When need to allocate:
  - If enough room in free space, use it and update free space pointer.
  - Else, try to compact, if successful, use the freed space.
  - Else, tell caller that page is full.
- Advantages:
  - Can move records around in page without changing their record ID
  - Allows lazy space management within the page, with opportunity for clean up later

Slotted page (continued)



- What’s the biggest record you can add to the above page?
  - Need 2 bytes for slot: [offset, length] plus record.
- What happens when a record needs to move to a different page?
  - Leave a “tombstone” behind, pointing to new page and slot.
  - Record id remains unchanged – no more than one hop needed.

So far we’ve organized:

- Fields into Records (fixed and variable length)
- Records into Pages (fixed and variable length)

Now we need to organize Pages into Files

Alternative File Organizations

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

Heap files: Unordered. Suitable when typical access is a file scan retrieving all records. Easy to maintain.

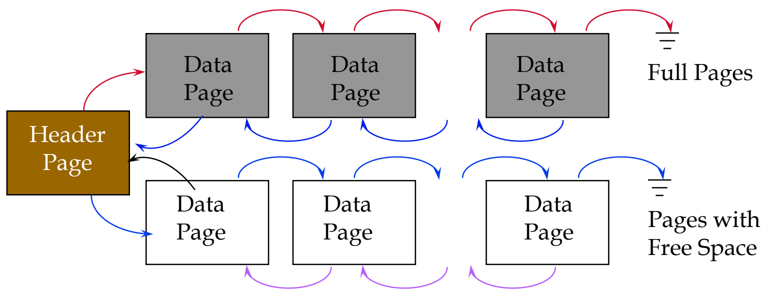
Sorted Files: Best for retrieval in *search key* order, or if only a `range' of records is needed. Expensive to maintain.

Clustered Files (with Indexes): A compromise between the above two extremes.

Unordered (Heap) Files

- Simplest file structure contains records in no particular order.
- As file grows and shrinks, pages are allocated and 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
- Can organize as a list, as a directory, a tree, …

Heap File Implemented as a List



- The Heap file name and header page id must be stored persistently.
  The catalog is a good place for this.
- Each page contains 2 `pointers' plus data.

Cost Model for Analysis

We ignore CPU costs, for simplicity:
- **B:** The number of data blocks
- **R:** Number of records per block
- **D:** (Average) time to read or write disk block

- Measuring number of block I/O's ignores gains of pre-fetching and sequential access; thus, even I/O cost is only loosely approximated.
- Average-case analysis; based on several simplistic assumptions.
  - Often called a "back of the envelope" calculation.
    ☛ *Good enough to show the overall trends!*

Some Assumptions in the Analysis

- Single record insert and delete.
- Equality selection - exactly one match (what if more or less???).
- For Heap Files we'll assume:
  - Insert always appends to end of file.
  - Delete just leaves free space in the page.
  - Empty pages are not deallocated.

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 File**
Scan all records	BD		
Equality Search (unique key)	0.5 BD		
Range Search	BD		
Insert	2D		
Delete	(0.5B+1)D		

Sorted Files

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

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 File**
Scan all records	BD	BD	
Equality Search (unique key)	0.5 BD	$(\log_2 B) * D$	
Range Search	BD	$[(\log_2 B)$ + #match pg]*D	
Insert	2D	$((\log_2 B)+B)D$ *(because rd,w0.5 File)*	
Delete	(0.5B+1)D	Same cost as Insert	