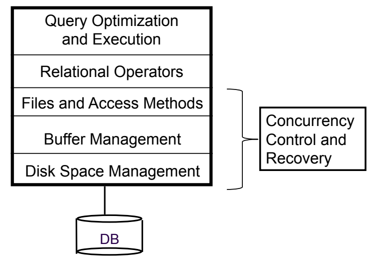
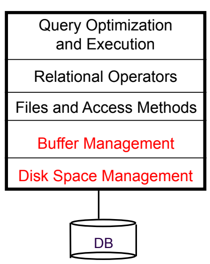
Storing Data: Disks and Files

Block diagram of a DBMS



Disks, Memory, and Files



A brief note on terminology

Block = Page
- Unit of transfer for disk read/write
- Typically 4KB in the book
  - And hence in formulas in slides
- 64KB is a good number today

Relation = Table

Tuple = Row = Record

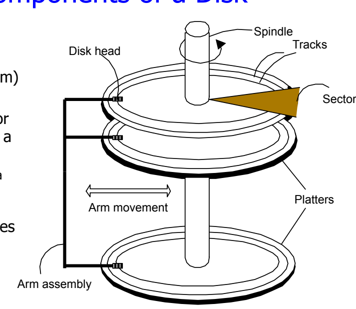
Attribute = Column = Field

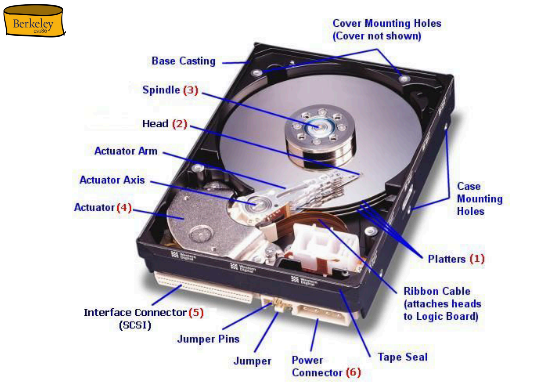
Disks and Files

- DBMS stores information on disks.
  - Disks are a mechanical anachronism!
- Major implications for DBMS design!
  - READ: transfer data from disk to main memory (RAM).
  - WRITE: transfer data from RAM to disk.
  - Both high-cost relative to memory references
    - Can/should plan carefully!

Recall: Components of a Disk

- Platters spin
  (say 120 rps = 7200 rpm)
- Arm assembly moved in or out to position a head on a desired track.
  - Tracks under heads make a *cylinder* (imaginary)
- Only one head reads/writes at any one time
- *page size* is a multiple of (fixed) *sector size*





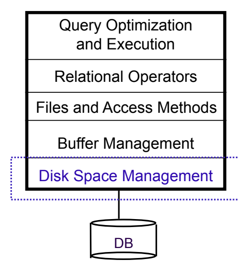
Recall: Accessing a Disk Page

- Time to access (read/write) a disk page:
  - *seek time* (moving arms to position disk head on track)
  - *rotational delay* (waiting for page to rotate under head)
  - *transfer time* (actually moving data to/from disk surface)
- Seek time and rotational delay dominate.
  - Seek time varies from 0 to 12 msec (average 4-9)
  - Rotational delay at 7200 rpm = 4.2 msec average
  - Transfer rate peak around 100 MB/sec
- Key to lower I/O cost: reduce seek/rotation delays! Hardware vs. software solutions?

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!

Context



Disk Space Management

- Lowest layer of DBMS, manages space on disk
- Higher levels call upon this layer to:
  - allocate/de-allocate a page
  - read/write a page
- Request for a *sequence* of pages best satisfied by pages stored sequentially on disk!
  - Responsibility of disk space manager.
  - Physical details hidden from higher levels of system
  - Though they may make performance assumptions!
    - Hence disk space manager should do a decent job.

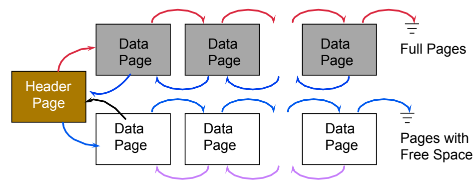
Files of Records

- Pages are the 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)
- Typically implemented as multiple OS "files"
  - Or "raw" disk space

Unordered (Heap) Files

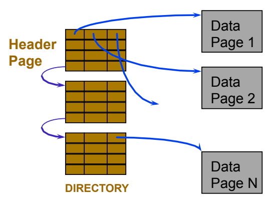
- Collection of records in no particular order.
- As file shrinks/grows, disk pages (de)allocated
- To support record level operations, we must:
  - keep track of the *pages* in a file
  - keep track of *free space* on pages
  - keep track of the *records* on a page
- There are many alternatives for keeping track of this.
  - We'll consider 2

Heap File Implemented as a List



- Header page ID and Heap file name stored elsewhere
  - Database "catalog"
- Each page contains 2 "pointers" plus data
  - Problem for multi-page objects (blobs) – how to read blobs?

Better: Use a Page Directory

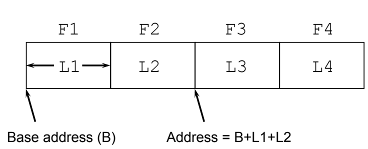


- Directory entries include #free bytes on the page.
- Directory is a collection of pages; linked list implementation is just one alternative.
  - *Much smaller than linked list of all HF pages*!
  - Can also point to groups of pages (say 64k chunks)

Indexes (sneak preview)

- A Heap file allows us to retrieve records:
  - by specifying the *rid*, or
  - by scanning all records sequentially
- Nice 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

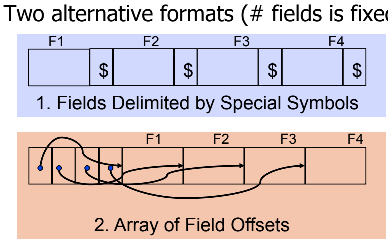
Record Formats: Fixed Length



- Field types same for all records in a file.
  - Type info stored separately in *system catalog.*
- Finding *i'th* field done via arithmetic like arrays

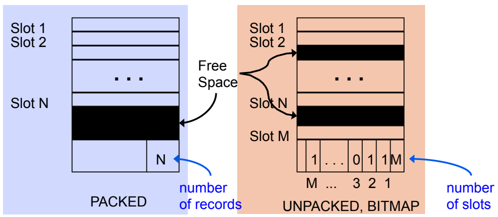
Record Formats: Variable Length

- Two alternative formats (# fields is fixed):



Second offers direct access to i'th field, efficient storage of *nulls* (special *unknown* value); small directory overhead.

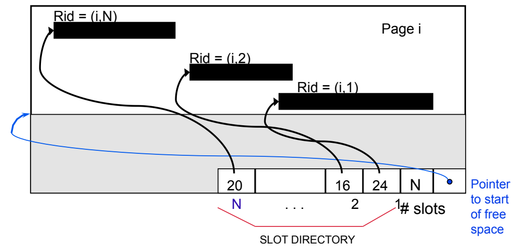
Page Formats: Fixed Length Records



Record id = <page id, slot #>.

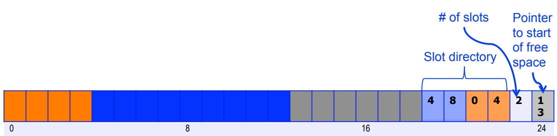
In first alternative, moving records for free space management *changes rid*; may be problematic!

"Slotted Page" Format: Variable Length Records



Can move records on page without changing rid!
So, attractive for fixed-length records too.

Slotted page: a detailed view



- Note each slot entry is (start, length)
- What's the biggest tuple you can add?
  - 2 bytes of slot space
  - *x* bytes of storage

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!*

Attr_Cat(attr_name, rel_name, type, position)

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

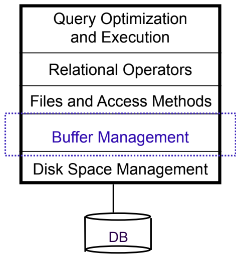
pg_attribute

cs186 — vagrant@precise64: ~ — ssh — 80×30

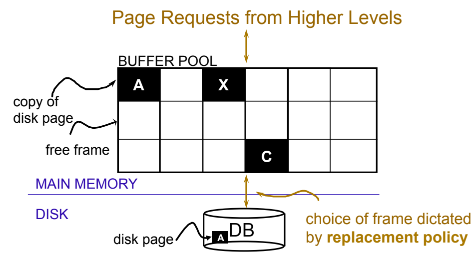
```
postgres=# \d pg_attribute
    Table "pg_catalog.pg_attribute"
    Column     |   Type    | Modifiers
---------------+-----------+-----------
 attrelid      | oid       | not null
 attname       | name      | not null
 atttypid      | oid       | not null
 attstattarget | integer   | not null
 attlen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

Context



Buffer Management in a DBMS



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

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!

More on Buffer Management

- Requestor of page must eventually:
  1. *unpin* it
  2. indicate whether page was modified via *dirty* bit.
- 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")
- Concurrency control & recovery may do additional I/Os upon replacement.
  - *Write-Ahead Logging* protocol; more later!

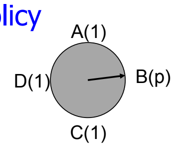
Buffer Replacement Policy

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

LRU Replacement Policy

- *Least Recently Used (LRU)*
  - (Frame pinned: "in use", not available to replace)
  - track time each frame last *unpinned* (end of use)
  - replace the frame which has the earliest unpinned time
- Very common policy: intuitive and simple
  - Works well for repeated accesses to popular pages
- *Problem: Sequential flooding*
  - LRU + repeated sequential scans.
  - # buffer frames < # pages in file? Each page request causes I/O!
  - What's better in this scenario?

"Clock" Replacement Policy



- An approximation of LRU
- Arrange frames into a (logical) 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 frame in cycle {
        if (pincount == 0 && ref bit is on)
                turn off ref bit; // 2nd chance
        else if (pincount == 0 && ref bit is off)
                choose this page for replacement;
} until a page is chosen;
```

DBMS vs. OS File System

OS does disk space & buffer mgmt:
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.

- I/O typically done via lower-level OS interfaces
  - Avoid OS "file cache"
  - Control write timing, prefetching

Summary

- Disks provide cheap, non-volatile storage.
  - Better random access than tape, worse than RAM
  - Magnetic disks well understood; flash evolving quickly.
  - For mag disk, arrange data to minimize *seek* and *rotation* delays.
    - Depends on workload!
- DBMS vs. OS File Support
  - DBMS needs non-default features
  - Careful timing of writes, control over prefetch
- Variable length record format
  - Direct access to i'th field and null values.
- Slotted page format
  - Variable length records and intra-page reorg

Summary (Contd.)

- DBMS "File" tracks collection of pages, records within each.
  - Pages with free space identified using linked list or directory structure

- Indexes support efficient retrieval of records based on the values in some fields.

- Catalog relations store information about relations, indexes and views.

Summary (Contd.)

- Buffer manager brings pages into RAM.
  - Page pinned in RAM until released by requestor.
  - Dirty pages written to disk when frame replaced (sometime after requestor unpins the page).
  - Choice of frame to replace based on *replacement policy.*
  - Tries to *pre-fetch* several pages at a time.