FILES (AND DISKS)
Block diagram of a DBMS

- Query Optimization and Execution
- Relational Operators
- Files and Access Methods
- Buffer Management
- Disk Space Management

DB
Disks, Memory, and Files

- Query Optimization and Execution
- Relational Operators
- Files and Access Methods
- Buffer Management
- Disk Space Management

DB
Disks and Files

• DBMS stores information on disks.
  – Disks are a mechanical devices!

• Major implications for DBMS design!
  – Unit of data transfer: a disk block (or page)
  – **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)

- 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

- Block size is a multiple of (fixed) sector size
Recall: Accessing a Disk Page

• Time to access (read/write) a disk block:
  – *seek time* (moving arms to position disk head on track)
  – *rotational delay* (waiting for block 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 10msec
  – Rotational delay varies from 0 to 3msec
  – Transfer rate around .02msec per 8K block

• Key to lower I/O cost: *reduce seek/rotation delays!* Hardware vs. software solutions?
Context

- Query Optimization and Execution
- Relational Operators
- Files and Access Methods
- Buffer Management
- Disk Space Management

DB
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.
  – Higher levels don’t know how this is done, or how free space is managed.
  – Though they may make performance assumptions!
    • Hence disk space manager should do a decent job.
Files of Records

• Blocks 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.
Heap File Implemented as a List

- The header page id and Heap file name must be stored someplace.
  - Database "catalog"
- Each page contains 2 `pointers’ plus data.
Better: Use a Page Directory

- The entry for a page can include the number of free bytes on the page.
- The directory is a collection of pages; linked list implementation is just one alternative.
  - Much smaller than linked list of all HF pages!
Sorted File

• Sort the records in a file ordered by the values of one of the attributes
  – Usually, we use the Primary key

• Can be packed or not..
Indexes!

- **Indexes**: file structures for efficient *value-based* queries
- Can be used for storing the records in pages
  - Indexed file organization!
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.

```
F1
L1

F2
L2

F3
L3

F4
L4
```

Base address (B)  
Address = B + L1 + L2
Record Formats: Variable Length

• Two alternative formats (# fields is fixed):

Second offers direct access to i’th field, efficient storage of nulls (special don’t know value); small directory overhead.
Record id = <page id, slot #>. In first alternative, moving records for free space management changes rid; may not be acceptable.
Page Formats: Variable Length Records

Can move records on page without changing rid; so, attractive for fixed-length records too.
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)

<table>
<thead>
<tr>
<th>attr_name</th>
<th>rel_name</th>
<th>type</th>
<th>position</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr_name</td>
<td>Attribute_Cat</td>
<td>string</td>
<td>1</td>
</tr>
<tr>
<td>rel_name</td>
<td>Attribute_Cat</td>
<td>string</td>
<td>2</td>
</tr>
<tr>
<td>type</td>
<td>Attribute_Cat</td>
<td>string</td>
<td>3</td>
</tr>
<tr>
<td>position</td>
<td>Attribute_Cat</td>
<td>integer</td>
<td>4</td>
</tr>
<tr>
<td>sid</td>
<td>Students</td>
<td>string</td>
<td>1</td>
</tr>
<tr>
<td>name</td>
<td>Students</td>
<td>string</td>
<td>2</td>
</tr>
<tr>
<td>login</td>
<td>Students</td>
<td>string</td>
<td>3</td>
</tr>
<tr>
<td>age</td>
<td>Students</td>
<td>integer</td>
<td>4</td>
</tr>
<tr>
<td>gpa</td>
<td>Students</td>
<td>real</td>
<td>5</td>
</tr>
<tr>
<td>fid</td>
<td>Faculty</td>
<td>string</td>
<td>1</td>
</tr>
<tr>
<td>fname</td>
<td>Faculty</td>
<td>string</td>
<td>2</td>
</tr>
<tr>
<td>sal</td>
<td>Faculty</td>
<td>real</td>
<td>3</td>
</tr>
</tbody>
</table>
Context

- Query Optimization and Execution
- Relational Operators
- Files and Access Methods
- Buffer Management
- Disk Space Management

DB
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**

Diagram:
- Page Requests from Higher Levels
- Buffer Pool
- Main Memory
- Disk
- Copy of disk page
- Free frame
- Choice of frame dictated by replacement policy
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”)

• CC & recovery may do additional I/Os upon replacement.
  – *Write-Ahead Log* 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 means each page request causes an I/O.
  – Idea: MRU better in this scenario?
“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:
```c
  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
  - 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.