Where is the data and how to get to it?
DBMS architecture

- Query Parser
- Query Rewriter
- Query Optimizer
- Query Executor

- Lock Manager
- Access Methods
- Buffer Manager
- Log Manager

- Disk Space Manager

- DB
Memory hierarchy

- **Main memory**: Random access, fast, volatile
- **Magnetic disk**: Random access, relatively slow, non-volatile
- **Tape**: Sequential scan, non-volatile, long archiving
Disks and DBMS design

Databases are stored on disks

expensive operations
Why not store everything in memory?

volatility

cost
Basics of disks

Platters spin under the head

Only one head reads and writes

Platters have tracks, forming an (imaginary) cylinder

Each track has sectors. Blocks (pages) are multiple of sectors

Retrieval time varies:
Seek time + rotation delay + transfer time
Accessing a disk page

- Time to access (read/write) a disk block:
  1. *seek time* (moving arms to position a disk head on a track)
  2. *rotational delay* (waiting for a block to rotate under the head)
  3. *transfer time* (actually moving data to/from disk surface)

- Seek time and rotational delay dominate.

- Placement of pages on disk has major impact on DBMS performance.
Arranging pages on disk

- **Sequential page storage:**
  - blocks on the same track, followed by
  - blocks on the same cylinder, followed by
  - blocks on an adjacent cylinder

- Pages in a *file* should be arranged sequentially on disk, to minimize seek and rotational delay.
  - Scan of the file is a *sequential scan*. 
Files of records

Fields are organized in a record

A collection of records are organized in a page

A collection of pages makes a file
Unordered (Heap) Files

- Simplest file structure contains records in no particular order.

- As file grows and shrinks, disk 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
Page entry can include the number of free bytes on the page.

The directory is a collection of pages; linked list implementation is just one alternative.
Page format

- How to store records on a page

- Consider a page as a collection of slots, one for each record

- A record is identified by rid = <page id, slot #>

- Record ids (rids) are used in indexes
Page formats: fixed length records

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.
Record formats: fixed length

Number of fields and type stored in *system catalogs*.

Finding $i^{th}$ field does not require scan of record.
Record formats: variable length

Fields Delimited by Special Symbols

Array of Field Offsets

2nd choice offers direct access to i\textsuperscript{th} field with small directory overhead.
Question

Consider the following query:

```
SELECT S1.temp, S2.pressure
FROM TempSensor S1, PressureSensor S2
WHERE S1.location = S2.location
AND S1.time = S2.time
```

How can the DBMS execute this query given

- 1 GB of memory
- 100 GB TempSensor and 10 GB PressureSensor
Buffer manager

Page requests from higher-level code

**Buffer pool**

**Disk page**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

**Free frame**

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Disk**

1 page corresponds to 1 disk block

**Disk = collection of blocks**

**Files and access methods**

**Buffer pool manager**

**Main memory**

choice of frame dictated by replacement policy

**Disk space manager**

**READ/WRITE**

**INPUT/OUTPUT**

• Data must be in RAM for DBMS to operate on it!
• Buffer pool = table of <frame#, pageid> pairs
When a page is requested...

- If requested page is not in pool (and buffer is full):
  - Choose a frame for *replacement*
  - If frame is dirty, write it to disk
  - Read requested page into chosen frame
- *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!
Buffer replacement policy

- Frame is chosen for replacement by a replacement policy:
  - Least-recently-used (LRU), Clock, MRU etc.
- Policy can have big impact on # of I/O’s; depends on the access pattern.
- **Sequential flooding**: Nasty situation caused by 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).
DBMS vs. OS file system

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

- **Reason 1: Correctness**
  - DBMS needs fine grained control for transactions
  - Needs to force pages to disk for recovery purposes

- **Reason 2: Performance**
  - DBMS may be able to anticipate access patterns
  - Hence, may also be able to perform prefetching
  - May select better page replacement policy
Database file types

The data file can be one of:

- **Heap file**
  - Set of records, partitioned into blocks
  - Unsorted

- **Sequential file**
  - Sorted according to some attribute(s) called *(sort) key*

**different from “key”!**
Index

- A (possibly separate) file, that allows fast access to records in the data file given a search key.
- The index contains (key, value) pairs:
  - The key = an attribute value
  - The value = either a pointer to the record, or the record itself again different from “key”!
High-level overview: Indexes

data file = index file
clustered (primary) index

index file
unclustered (secondary) index
Index classification

- **Clustered/unclustered**
  - Clustered = records close in index are close in data
  - Unclustered = records close in index may be far in data

- **Primary/secondary**
  - Primary = is over attributes that include the primary key
  - Secondary = otherwise

- **Organization:** B+ tree or Hash table
Clustered/Unclustered

- **Clustered**
  - Index determines the location of indexed records
  - Typically, clustered index is one where values are data records (but not necessary)

- **Unclustered**
  - Index cannot reorder data, does not determine data location
  - In these indexes: value = pointer to data record
Clustered index

- File is sorted on the index attribute
- Only one per table
Unclustered index

- Several per table
Clustered vs. unclustered index

CLUSTERED

UNCLUSTERED
Alternatives for data entry $k^*$ in index

- In a data entry $k^*$, we can store:
  - Alternative 1: $<k, \text{data record with search key value } k>$
  - Alternative 2: $<k, \text{rid of a record with search key value } k>$
  - Alternative 3: $<k, \text{list of rids of records with search key } k>$

- Choice of an alternative for data entries is orthogonal to an indexing technique used.
  - Indexing techniques: B+ tree, hashing, ...
Cost model

We ignore CPU costs, for simplicity:

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

Measuring number of page I/Os ignores gains of pre-fetching a sequence of pages; thus, even I/O cost is only approximated.

Average-case analysis; based on several simplistic assumptions.
Comparing file organizations

- Heap files (random order)
- Sorted files, sorted on \(<age, sal>\)
- Clustered B+ tree file, Alternative (1), search key \(<age, sal>\)
- Heap file with unclustered B + tree index on search key \(<age, sal>\)
- Heap file with unclustered hash index on search key \(<age, sal>\)
Operations to compare

- Scan: Fetch all records from disk
- Equality search
- Range selection
- Insert a record
- Delete a record
Assumptions

- **Heap Files:**
  - Equality selection on key; exactly one match.

- **Sorted Files:**
  - Files compacted after deletions.

- **Indexes:**
  - Alt (2), (3): data entry size = 10% size of record
  - Hash: No overflow buckets.
    - 80% page occupancy => File size = 1.25 data size
  - Tree: 67% occupancy (this is typical).
    - Implies file size = 1.5 data size
Assumptions (contd.)

- **Scans:**
  - Leaf levels of a tree-index are chained.
  - Index data-entries plus actual file scanned for unclustered indexes.

- **Range searches:**
  - We use tree indexes to restrict the set of data records fetched, but ignore hash indexes.
# Cost of operations

<table>
<thead>
<tr>
<th></th>
<th>Scan</th>
<th>Equality</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Heap file</strong></td>
<td>BD</td>
<td>0.5 BD</td>
<td>BD</td>
</tr>
<tr>
<td><strong>Sorted file</strong></td>
<td>BD</td>
<td>D log&lt;sub&gt;2&lt;/sub&gt; B</td>
<td>D (log&lt;sub&gt;2&lt;/sub&gt; B + #match recs)</td>
</tr>
<tr>
<td><strong>Clustered tree index</strong></td>
<td>1.5 BD</td>
<td>D log&lt;sub&gt;F&lt;/sub&gt; 1.5B</td>
<td>D (log&lt;sub&gt;F&lt;/sub&gt; 1.5B + #pages with matched recs)</td>
</tr>
<tr>
<td><strong>Unclustered tree index</strong></td>
<td>BD (R+0.15)</td>
<td>D(1 + log&lt;sub&gt;F&lt;/sub&gt; 0.15B)</td>
<td>D (log&lt;sub&gt;F&lt;/sub&gt; 0.15B + #pages with matched recs)</td>
</tr>
<tr>
<td><strong>Unclustered hash index</strong></td>
<td>BD (R + 0.125)</td>
<td>2D</td>
<td>BD</td>
</tr>
</tbody>
</table>

Several assumptions underlie these (rough) estimates!