Storing Data: Disks and Files

“Yea, from the table of my memory
I’ll wipe away all trivial fond records.”
-- Shakespeare, *Hamlet*
Data Access
DBMS stores information on ("hard") disks.
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 Everything in Main Memory?

- **Costs too much.** $xxxx will buy you either xxxMB of RAM or xxxGB of disk.

- **Main memory is volatile.** We want data to be saved between runs. (Obviously!)

- Typical storage hierarchy:
  - Main memory (RAM) for currently used data.
  - Disk for the main database (secondary storage).
  - Tapes for archiving older versions of the data (tertiary storage).
Disks

- Secondary storage device of choice.
- Main advantage over tapes: \textit{random access} vs. \textit{sequential}.
- Data is stored and retrieved in units called \textit{disk blocks} or \textit{pages}.
- Unlike RAM, time to retrieve a disk page varies depending upon location on disk.
  - Therefore, relative placement of pages on disk has major impact on DBMS performance!
Components of a Disk

- The platters spin (say, 7200rpm).

- The arm assembly is 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 sector size (which is fixed).
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 about 1 to 20msec
  - Rotational delay varies from 0 to 10msec
  - Transfer rate is about 1msec per 4KB page

- **Key to lower I/O cost:** reduce seek/rotation delays!

- **Hardware vs. software solutions?**
Arranging Pages on Disk

- 'Next' block concept:
  - blocks on same track, followed by
  - blocks on same cylinder, followed by
  - blocks on adjacent cylinder

- Blocks in a file should be arranged sequentially on disk (by `next`), to minimize seek and rotational delay.

- For a sequential scan, **pre-fetching** several pages at a time is a big win!
RAID

- Disk Array: Arrangement of several disks that gives abstraction of a single, large disk.
- Goals: Increase performance and reliability.
Terms

- Redundancy – multiple copies of blocks/partitions
- Mirror – a complete copy of a drive
- Data striping – data is segmented into equal-size partitions that are distributed over multiple disks
- Parity – an extra check disk is contains information that can be used to recover from failure of any one disk in the array.
RAID (Redundant Array of Inexpensive or Independent Disks) is an important component for servers on a critical enterprise or workgroup network.

- RAID provides crash-proof hard drive systems.
RAID Background

- RAID as a computer concept has been around for over twenty years. The computer science department at UC Berkeley first developed the RAID concept back in the 1980's.
- Used the word *Inexpensive* rather than today's *independent*.
- RAID systems not only increase reliability, they also increase available storage capacity. RAID systems have over 1,000 Gigabyte or 1 Terabyte.
RAID Levels

- Six distinctive RAID levels have been developed and agreed upon, voluntarily, by various manufacturers. These RAID levels are 0, 1, 2, 3, 4, and 5. Other combinations of these levels are also used, such as level 10 (which is 0+1) or level 6 (which is 5+1).

- A RAID system appears as a single large hard disk to the operating system.

- All of the computations associated with creating the RAID set are hidden from the operating system.

- RAID responds to standard disk commands such as read, write, and format.
- **RAID Level 0** stripes data across all disks without redundancy or parity.
- This Level maximizes data transfer rates and is good for handling large files.
RAID Level 1 mirrors data across multiple disks. Data is duplicated on another set of drives. If one drive fails, then the data is still available on the other mirror.

This Level has the highest cost per MB and is best suited for smaller capacity applications such as mirroring the boot drive.

Typically only one drive is mirrored at a time.
RAID Level 2 bit interleaves data across multiple disks with parity information created using a Hamming code.

A Hamming code detects errors that occur and determines which part is in error. RAID Level 2 specifies 39 disks with 32 disks of user storage and 7 disks of error recovery coding.

This Level is not used much.
- **RAID Levels 3 and 4** stripe data across multiple drives and write parity to a dedicated drive.
- Level 3 is typically implemented at the BYTE level. While Level 4 is typically implemented at the BLOCK level.
- These Levels combine the performance of RAID 0 with a redundancy feature.
- If a drive fails, the data can be restructured by the parity drive. RAID 3 and 4 are best suited for large transfer sizes and rates where redundancy is important.
- The parity information is calculated during write time and can effect overall performance.
- **RAID Level 5** stripes data and parity information at the block level across all the drives in the array.
- Parity is written onto the next available drive rather than a dedicated parity drive.
- Reads and writes may be performed concurrently.
- Spare drives take over in the event of a drive failure.
“How index-learning turns no student pale
Yet holds the eel of science by the tail.”
-- Alexander Pope (1688-1744)
Many alternatives exist, each ideal for some situation, and not so good in others:

- **Heap files:**
  - Suitable when typical access is a file scan retrieving all records.
  - A file organization for a relation in which new tuples are added at the end of the file, with new pages allocated there as needed.
  - The pages may or may not be physically contiguous on disk, but performance is best if they are.
  - Tuple deletion can result in file fragmentation over time, so periodic reorganization is necessary to maintain performance and space utilization.
  - This is the most common default file organization in relational products and is the only option in some (indices are used to achieve other organizations).
**Sorted Files:**

- Arranging records in a file according to a specified sequence, such as alphabetically or numerically, from lowest to highest.
- Best if records must be retrieved in some order, or only a `range` of records is needed.
Hashed Files:

- We are assuming that a record in a database always has associated with it
  - (a) an address on the disk or in memory,
  - (b) a key (either external to data or some aspect of that data).
- Hashing consists of using an algorithm to compute a record's address from its key.
- When a hash file is created, the records are placed at the address obtained by applying the hashing algorithm to the record's key.
- The same hashing algorithm is used to retrieve a record given its key.
An *index* on a file speeds up selections on the *search key fields* for the index.

- Any subset of the fields of a relation can be the search key for an index on the relation.
- *Search key* is not the same as *key* (minimal set of fields that uniquely identify a record in a relation).

An index contains a collection of *data entries*, and supports efficient retrieval of all data entries with a given key value.

Three alternatives:

1. Search key value
2. Record id of data record with search key value
3. Record id list of data records with search key value
Alternatives

- **Alternative 1:** *Search key value*
  - Index structure is a file organization for data records (like Heap files or sorted files).
  - If data records very large, # of pages containing data entries is high.
  - Implies size of auxiliary information in the index is also large, typically.
Alternatives (Contd.)

- **Alternatives 2 and 3:** *Record id of data record with search key value* and *Record id list of data records with search key*

- Data entries typically much smaller than data records. So, better than Alternative 1 with large data records, especially if search keys are small.
  - If more than one index is required on a given file, at most one index can use Alternative 1; rest must use Alternatives 2 or 3.
  - Alternative 3 more compact than Alternative 2, but leads to variable sized data entries even if search keys are of fixed length.
Index Classification

- **Primary vs. secondary.** If search key contains primary key, then called primary index.

- **Clustered vs. unclustered:** If order of data records is the same as, or `close to`, order of data entries, then called clustered index.
  - Alternative 1 implies clustered
  - A file can be clustered on at most one search key.
  - Cost of retrieving data records through index varies *greatly* based on whether index is clustered or not!
Clustered vs. Unclustered Index

- Suppose that Alternative (2) is used for data entries, and that the data records are stored in a Heap file.
  - To build clustered index, first sort the Heap file.
  - Overflow pages may be needed for inserts.
Index Classification (Contd.)

- **Dense vs. Sparse:** If there is at least one data entry per search key value, then dense.
  - Alternative 1 always leads to dense index.
  - Sparse indexes are smaller; however, some useful optimizations are based on dense indexes.
Trees

- A tree imposes a hierarchical structure on a collection of items e.g., genealogies, organization charts

- Used long before DBMS
- However, arise naturally in many different areas of computer science: where?
A tree is a collection of elements called *nodes*, one of which is distinguished as the *root*.

- Relationship ("parenthood") which places hierarchical structure on the nodes.
- Node can be of whatever type we wish.
Recursive Definition

1. A single node by itself is a tree. This node is also the root of the tree.

2. Suppose $n$ is a node and $T_1, T_2, ..., T_k$ are trees with roots $n_1, n_2, ..., n_k$ respectively. We can construct a new tree by making $n$ be the parent of nodes $n_1, n_2, ..., n_k$. In this tree $n$ is the root and $T_1, T_2, ..., T_k$ are the subtrees of the root. Nodes $n_1, n_2, ..., n_k$ are called the children of node $n$.

The null tree is a “tree” with no nodes, represented by $\Lambda$. 
A binary tree is either:

1. an empty tree, or
2. a tree in which every node has either no children, a left child, a right child, or both a left and a right child.

- Each element in a binary tree has exactly two subtrees (one or both may be empty binary trees).
- The subtrees of each element in a binary tree are ordered (distinguish between right and left subtrees).
Examples of Binary Trees

# nodes = 9
height of root = 4
Properties of Binary Trees

1. A binary tree with \( n \) nodes has \( n-1 \) edges
2. A binary tree of height \( h, h \geq 0 \), has at least \( h \) and at most \( 2^h-1 \) elements in it
3. The height of a binary tree that contains \( n \) elements is at most \( n \) and at least \( \lceil \log_2(n+1) \rceil \)
Motivation

- When data is too large to fit in main memory, then the number of disk accesses becomes important.
- A disk access is unbelievably expensive compared to a typical computer instruction (mechanical limitations).
- One disk access is worth about 200,000 instructions.
- The number of disk accesses will dominate the running time.
Motivation (cont.)

- Secondary memory (disk) is divided into equal-sized **blocks** (typical sizes are 512, 2048, 4096 or 8192 bytes)
- The basic I/O operation transfers the contents of one disk block to/from main memory.
- Our goal is to devise a **multiway search tree** that will minimize file accesses (by exploiting disk block read).
Definition of a B-Tree

A B-tree of order \( m \) is an \( m \)-way tree such that

- All leaves are on the same level.
- All internal nodes except the root are constrained to have at most \( m \) non empty children and at least \( m/2 \) non empty children.
- The root has at most \( m \) non empty children.
B+ Trees

- The previous discussion mentioned that B-Trees are universally used to implement large-scale disk-based file systems.
- What is implemented is a variant of the B-tree called B+ tree.
- The most significant difference between a B+ and B trees is that B+ trees store records only at the leaf nodes. The internal nodes only store key values to help guide search.
- The big advantage of B+ trees is that the keys are small enough so that the top levels of the tree can remain in main memory and do not require any disk accesses.