COSC 404
Database System Implementation
Data Storage and Organization

Dr. Ramon Lawrence
University of British Columbia Okanagan
ramon.lawrence@ubc.ca

Storage and Organization Overview

The first task in building a database system is determining how to represent and store the data.

Since a database is an application that is running on an operating system, the database must use the file system provided by the operating system to store its information.

- However, many database systems implement their own file security and organization on top of the operating system file structure.

We will study techniques for storing and representing data.

Representing Data on Devices

Physical storage of data is dependent on the computer system and its associated devices on which the data is stored.

How we represent and manipulate the data is affected by the physical media and its properties.

- sequential versus random access
- read and write costs
- temporary versus permanent memory

Review: Memory Definitions

Temporary memory retains data only while the power is on.

- Also referred to as ***volatile storage***.
- e.g. dynamic random-access memory (DRAM) (main memory)

Permanent memory stores data even after the power is off.

- Also referred to as ***non-volatile storage***.
- e.g. flash memory, hard drive, SSD, DVD, tape drives
- Most permanent memory is **secondary storage** because the memory is stored in a separate device such as a hard drive.

Cache is faster memory used to store a subset of a larger, slower memory for performance.

- processor cache (Level 1 & 2), disk cache, network cache

Research Question In-Memory Database

Question: Does an in-memory database need a secondary storage device for persistence?

A) Yes

B) No

Review: Sequential vs. Random Access

RAM, hard drives, and flash memory allow random access. ***Random access*** allows retrieval of any data location in any order.

Tape drives allow sequential access. ***Sequential access*** requires visiting all previous locations in sequential order to retrieve a given location.

- That is, you cannot skip ahead, but must go through the tape in order until you reach the desired location.

COSC 404 - Dr. Ramon Lawrence

Review: Memory Sizes

Memory size is a measure of memory storage capacity.

- Memory size is measured in ***bytes***.
  - Each byte contains 8 **bits** - a bit is either a 0 or a 1.
  - A byte can store one character of text.
- Large memory sizes are measured in:
  - kilobytes (KBs) = 10^3 = 1,000 bytes
  - kibibyte (KiB) = 2^{10} = 1,024 bytes
  - megabytes (MBs) = 10^6 = 1,000,000 bytes
  - mebibyte (MiBs) = 2^{20} = 1,048,576 bytes
  - gigabytes (GBs) = 10^9 = 1,000,000,000 bytes
  - gibibytes (GiBs) = 2^{30} = 1,073,741,824 bytes
  - terabytes (TBs) = 10^{12} = 1,000,000,000,000 bytes
  - tebibytes (TiBs) = 2^{40} = 1,099,511,627,776 bytes

Page 7

COSC 404 - Dr. Ramon Lawrence

Transfer Size, Latency, and Bandwidth

Transfer size is the unit of memory that can be individually accessed, read and written.

- DRAM, EEPROM – byte addressable
- Hard drive, flash – block addressable (must read/write blocks)

Latency is the time it takes for information to be delivered after the initial request is made.

Bandwidth is the rate at which information can be delivered.

- ***Raw device bandwidth*** is the maximum sustained transfer rate of the device to the interface controller.
- ***Interface bandwidth*** is the maximum sustained transfer rate of the interface device onto the system bus.

Page 8

COSC 404 - Dr. Ramon Lawrence

Memory Devices Dynamic Random Access Memory

Dynamic random access memory (DRAM) is general purpose, volatile memory currently used in computers.

- DRAM uses only one transistor and one capacitor per bit.
- DRAM needs periodic refreshing of the capacitor.

DRAM properties:

- low cost, high capacity
- volatile
- byte addressable
- latency ~ 10 ns
- bandwidth = 5 to 20 GB/s

Page 9

COSC 404 - Dr. Ramon Lawrence

Memory Devices Processor Cache

Processor cache is faster memory storing recently used data that reduces the average memory access time.

- Cache is organized into lines/blocks of size from 64-512 bytes.
- Various levels of cache with different performance.

Cache properties:

- higher cost, very low capacity
- cache operation is hardware controlled
- byte addressable
- latency – a few clock cycles
- bandwidth – very high, limited by processor bus

Page 10

COSC 404 - Dr. Ramon Lawrence

Memory Devices Flash Memory

Flash memory is used in many portable devices (cell phones, music/video players) and also solid-state drives.

NAND Flash Memory properties:

- non-volatile
- low cost, high capacity
- block addressable
- asymmetric read/write performance: reads are fast, writes (which involve an erase) are slow
- erase limit of 1,000,000 cycles
- bandwidth (per chip): 40 MB/s (read), 20 MB/s (write)

Page 11

COSC 404 - Dr. Ramon Lawrence

Memory Devices EEPROM

EEPROM (**E**lectrically **E**rasable **P**rogrammable **R**ead-**O**nly **M**emory) is non-volatile and stores small amounts of data.

- Often available on small microprocessors.

EEPROM properties:

- non-volatile
- high cost, low capacity
- byte addressable
- erase limit of 1,000,000 cycles
- latency: 250 ns

Page 12

Memory Devices
Magnetic Tapes

Tape storage is non-volatile and is used primarily for backup and archiving data.

- Tapes are ***sequential access*** devices, so they are much slower than disks.

Since most databases can be stored in hard drives and RAID systems that support direct access, tape drives are now relegated to secondary roles as backup devices.

- Database systems no longer worry about optimizing queries for data stored on tapes.

"**Tape is Dead. Disk is Tape. Flash is Disk. RAM Locality is King**." – Jim Gray (2006), Microsoft/IBM, Turing Award Winner 1998 - For seminal contributions to database and transaction processing research and technical leadership in system implementation.

Memory Devices
Solid State Drives

A ***solid state drive*** uses flash memory for storage.

Solid state drives have many benefits over hard drives:

- Increased performance (especially random reads)
- Better power utilization
- Higher reliability (no moving parts)

The performance of the solid state drive depends as much on the drive organization/controller as the underlying flash chips.

- Write performance is an issue and there is a large erase cost.

Solid state drives are non-volatile and block addressable like hard drives. The major difference is random reads are much faster (no seek time). This has a dramatic affect on the database algorithms used, and it is an active research topic.

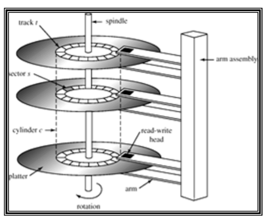
☆ Memory Devices
Hard Drives

Data is stored on a ***hard drive*** on the surface of ***platters***. Each ***platter*** is divided into circular tracks, and each ***track*** is divided into sectors. A ***sector*** is the smallest unit of data that can be read or written. A ***cylinder i*** consists of the ***i***-th track of all the platters (surfaces).

The ***read-write head*** is positioned close to the platter surface where it reads/writes magnetically encoded data.

To read a sector, the head is moved over the correct track by the ***arm assembly***. Since the platter spins continuously, the head reads the data when the sector rotates under the head.

Head-disk assemblies allow multiple disk platters on a single spindle with multiple heads (one per platter) mounted on a common arm.



Disk Controller and Interface

The ***disk controller*** interfaces between the computer system and the disk drive hardware.

- Accepts high-level commands to read or write a sector.
- Initiates actions such as moving the disk arm to the right track and actually reading or writing the data.
- Uses a data buffer and will re-order requests for increased performance.

The disk controller has the interface to the computer.

- E.g. 3.0 Gbit/s SATA can transfer from disk buffer to computer at 300 MB/s. Note that 7200 RPM disk has a sustained disk-to-buffer transfer rate of only about 70 MB/sec.

Device Performance Calculations

We will use simple models of devices to help understand the performance benefits and trade-offs.

These models are simplistic yet provide metrics to help determine when to use particular devices and their performance.

Memory Performance Calculations

Memory model will consider only transfer rate (determined from bus and memory speed). We will assume sequential and random transfer rates are the same.

Limitations:

- There is an advantage to sequential access compared to completely random access, especially with caching. Cache locality has a major impact as can avoid accessing memory.
- Memory alignment (4 byte/8 byte) matters.
- Memory and bus is shared by multiple processes.

Memory Performance Calculations Example

A system has 8 GB DDR4 memory with 20 GB/sec. bandwidth.

Question 1: How long does it take to transfer 1 contiguous block of 100 MB memory?

transfer time = 100 MB / 20,000 MB/sec. = 0.005 sec = **5 ms**

Question 2: How long does it take to transfer 1000 contiguous blocks of 100 KB memory?

transfer time = 1000 * (100 KB / 20,000,000 KB/sec.)
= 0.005 sec = **5 ms**

Disk Performance Measures

Disk capacity is the size of the hard drive.

- = #cylinders * #tracks/cylinder * #sectors/track * #bytes/sector

Disk access time is the time required to transfer data.

- = seek time + rotational latency + transfer time
- ***Seek time*** – time to reposition the arm over the correct track.
  - Average is 1/3rd the worst. (depends on arm position and target track)
- ***Rotational latency*** – time for first sector to appear under head.
  - Average latency is 1/2 of worst case. (one half rotation of disk)
- ***Transfer time*** – time to transfer data to memory.

Data-transfer rate – the rate at which data can be retrieved from disk which is directly related to the rotational speed.

Mean time to failure (MTTF) – the average time the disk is expected to run continuously without any failure.

Disk Performance Example

Given a hard drive with 10,000 cylinders, 10 tracks/cylinder, 60 sectors/track, and 500 bytes/sector, calculate its capacity.

Answer:

capacity = 10000 * 10 * 60 * 500 = 3,000,000,000 bytes
= 3,000,000,000 bytes / 1,048,576 bytes/MiB
= **2,861 MiB** = **2.8 GiB**
= **3,000 MB** = **3 GB**

Disk Performance Example (2)

If the hard drive spins at 7,200 rpm and has an average seek time of 10 ms, how long does a 2,000 byte transfer take?

Answer:

transfer size = 2,000 bytes / 500 bytes/sector = 4 sectors
revolution time = 1 / (7200 rpm / 60 rpm/sec) = 8.33 ms
latency = 1/2 revolution time on average = 4.17 ms
transfer time = revolution time * #sectorsTransfered / #sectors/track
= 8.33 ms * 4 / 60 = 0.56 ms
total transfer time = seek time + latency + transfer time
= 10 ms + 4.17 ms + 0.56 ms = **14.73 ms**

Sequential versus Random Disk Performance Example

A hard drive spins at 7,200 rpm, has an average seek time of 10 ms, and a track-to-track seek time of 2 ms. How long does a 1 MiB transfer take under the following conditions?

- Assume 512 bytes/sector, 64 sectors/track, and 1 track/cyl.

1) The data is stored randomly on the disk.

transfer size = 1,048,576 bytes / 512 bytes/sector = 2048 sectors
revolution time = 1 / (7200 rpm / 60 rpm/sec) = 8.33 ms
latency = 1/2 revolution time on average = 4.17 ms
transfer time = revolution time / #sectors/track
= 8.33 ms / 64 = 0.13 ms per sector
total transfer time = (seek time + latency + transfer time) * #sectors
= (10 ms + 4.17 ms + 0.13 ms)*2048
= **29,286.4 ms = 29.3 seconds**

Sequential versus Random Disk Performance Example (2)

2) The data is stored sequentially on the disk .

transfer size = 1,048,576 bytes / 512 bytes/sector = 2048 sectors
= 2048 sectors / 64 sectors/track = 32 tracks
latency = 1/2 revolution time on average = 4.17 ms
transfer time = revolution time / #sectors/track
= 8.33 ms / 64 = 0.13 ms per sector
total transfer time = seek time + latency + transfer time * #sectors + track-to-track seek time * (#tracks-1)
= 10 ms + 4.17 ms + 0.13 ms*2048 + 2 ms * 31
= **342.41 ms = 0.34 seconds**

3) What would be the optimal configuration of data if the hard drive had 4 heads? What is the time in this case?

Disk Performance Practice Questions

A Seagate Cheetah 15K 3.5" hard drive has 8 heads, 50,000 cylinders, 3,000 sectors/track, and 512 bytes/sector. Its average seek time is 3.4 ms with a speed of 15,000 rpm, and a reported data transfer rate of 600 MB/sec on a 6-Gb/S SAS interface.

1) What is the capacity of the drive?

2) What is the latency of the drive?

3) What is the maximum sustained transfer rate?

4) What is the total access time to transfer 400KiB?

Disk Performance Practice Questions Older Drive

The Maxtor DiamondMax 80 has 34,741 cylinders, 4 platters, each with 2 heads, 576 sectors/track, and 512 bytes/sector. Its average seek time is 9 ms with a speed of 5,400 rpm, and a reported maximum interface data transfer rate of 100 MB/sec.

1) What is the capacity of the Maxtor Drive?

2) What is the latency of the drive?

3) What is the actual maximum sustained transfer rate?

4) What is the total access time to transfer 4KB?

Hard Drive Model Limitations and Notes

- 1) Disk sizes are quoted after formatting.
  - ***Formatting*** is done by the OS to divide the disk into blocks.
  - A sector is a physical unit of the disk while a block is a logical OS unit.
- 2) Blocks are non-continuous. ***Interblock gaps*** store control information and are used to find the correct block on a track.
  - Since these gaps do not contain user data, the actual transfer rate is less than the theoretical transfer rate based on the rotation of the disk.
  - Manufactures quote bulk transfer rates (BTR) that measure the performance of reading multiple adjacent blocks when taking gaps into account. BTR = B/(B+G) * TR (B-block size, G-gap size)
- 3) Although the bit density on the media is relatively consistent, the number of sectors per track is not.
  - More sectors/track for tracks near outer edge of platter.
  - Faster transfer speed when reading outer tracks.
- 4) Buffering and read-ahead at controller and re-ordering requests (elevator algorithm) used to increase performance.

SSD Performance Calculations

SSD model will consider:

- **IOPS** – Input/Output Operations per Second (of given data size)
- latency
- bandwidth or transfer rate
- Different performance for read and write operations.

Limitations:

- Write bandwidth is not constant. It depends on request ordering and volume, space left in hard drive, and SSD controller implementation.

SSD Performance Calculations Examples

Question 1: A SSD has read bandwidth of 500 MB/sec. How long does it take to read 100 MB of data?

read time = 100 MB / 500 MB/sec. = **0.2 sec**

Question 2: The SSD IOPS for 4 KB write requests is 25,000. What is its effective write bandwidth?

write bandwidth = 25,000 IOPS * 4 KB requests

= 100,000 KB/sec. = **100 MB/sec.**

Device Performance

Question: What device would be the fastest to read 1 MB of data?

A) DRAM with bandwidth of 20 MB/sec.

B) SSD with read 400 IOPS for 100 KB data chunks.

C) 7200 rpm hard drive with seek time of 8 ms. Assume all data is on one track.

Summary of Memory Devices

Memory Type	Volatile?	Capacity	Latency	Bandwidth	Transfer Size	Notes
DRAM	yes	High	Small	High	Byte	Best price/speed.
Cache	Yes	Low	Lowest	Very high	Byte	Large reduction in memory latency.
NAND Flash	No	Very high	Small	High	Block	Asymmetric read/write costs.
EEPROM	No	Very low	Very small	High	Byte	High cost per bit. On small CPUs.
Tape Drive	No	Very high	Very high	Medium	Block	Sequential access: Even lost backup?
Solid State Drive	No	Very high	High	Medium	Block	Great random I/O. Issue in write costs.
Hard drive	No	Very high	High	Medium	block	Beats SSDs by cost/bit but not by performance/cost.

RAID

Redundant Arrays of Independent Disks is a disk organization technique that utilizes a large number of inexpensive, mass-market disks to provide increased reliability, performance, and storage.

- Originally, the "I" stood for inexpensive as RAID systems were a cost-effective alternative to large, expensive disks. However, now performance and reliability are the two major factors.

Improvement of Reliability via Redundancy

RAID systems improve reliability by introducing ***redundancy*** to the system as they store extra information that can be used to rebuild information lost due to a disk failure.

- Redundancy occurs by duplicating data across multiple disks.
- ***Mirroring*** or ***shadowing*** duplicates an entire disk on another. Every write is performed on both disks, and if either disk fails, the other contains all the data.

By introducing more disks to the system the chance that some disk out of a set of ***N*** disks will fail is much higher than the chance that a specific single disk will fail.

- E.g., A system with 100 disks, each with MTTF of 100,000 hours (approx. 11 years), will have a system MTTF of 1000 hours (approx. 41 days).

Review: Parity

Parity is used for error checking. A parity bit is an extra bit added to the data. A single parity bit can detect one bit error.

In ***odd parity*** the number of `1` bits in the data plus the parity bit must be odd. In ***even parity***, the number of `1` bits is even.

Example: What is the parity bit with *even parity* and the bit string: `01010010`?

- Answer: The parity bit must be a `1`, so that the # of `1`'s is even.

Parity Question

Question: What is the parity bit with *odd parity* and the bit string: `11111110`?

A) 0
B) 1
C) 2

Improvement in Performance via Parallelism

The other advantage of RAID systems is increased ***parallelism***. With multiple disks, two types of parallelism are possible:

- 1. Load balance multiple small accesses to increase throughput.
- 2. Parallelize large accesses to reduce response time.

Maximum transfer rates can be increased by allocating (***striping***) data across multiple disks then retrieving the data in parallel from the disks.

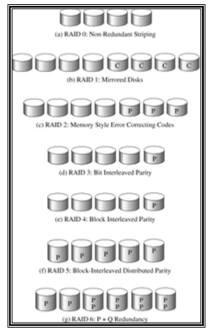
- ***Bit-level striping*** – split the bits of each byte across the disks
  - In an array of eight disks, write bit *i* of each byte to disk *i*.
  - Each access can read data at eight times the rate of a single disk.
  - But seek/access time worse than for a single disk.
- ***Block-level striping*** – with *n* disks, block *i* of a file goes to disk (*i* mod *n*) + 1

RAID Levels

There are different RAID organizations, or ***RAID levels***, that have differing cost, performance and reliability characteristics:

- **Level 0:** Striping at the block level (non-redundant).
- **Level 1:** Mirrored disks (redundancy)
- **Level 2:** Memory-Style Error-Correcting-Codes with bit striping.
- **Level 3:** Bit-Interleaved Parity - a single parity bit used for error correction. Subsumes Level 2 (same benefits at a lower cost).
- **Level 4:** Block-Interleaved Parity - uses block-level striping, and keeps all parity blocks on a single disk (for all other disks).
- **Level 5:** Block-Interleaved Distributed Parity - partitions data and parity among all *N* + 1 disks, rather than storing data in *N* disks and parity in 1 disk. Subsumes Level 4.
- **Level 6**: P+Q Redundancy scheme - similar to Level 5, but stores extra info to guard against multiple disk failures.

RAID Levels Discussion



Level 0 is used for high-performance where data loss is not critical (parallelism).

Level 1 is for applications that require redundancy (protection from disk failures) with minimum cost.

- Level 1 requires at least two disks.

Level 5 is a common because it offers both reliability and increased performance.

- With 3 disks, the parity block for *n*th block is stored on disk (*n mod* 3) + 1. Do not have single disk bottleneck like Level 4.

Level 6 offers extra redundancy compared to Level 5 and is used to deal with multiple drive failures.

RAID Question

Question: What RAID level offers the high performance but no redundancy?

A) RAID 0
B) RAID 1
C) RAID 5
D) RAID 6

RAID Practice Question

Question: The capacity of a hard drive is 800 GB. Determine the capacity of the following RAID configurations:

i) 8 drives in RAID 0 configuration
ii) 8 drives in RAID 1 configuration
iii) 8 drives in RAID 5 configuration

A) i) 6400 GB ii) 3200 GB iii) 5600 GB
B) i) 3200 GB ii) 6400 GB iii) 5600 GB
C) i) 6400 GB ii) 3200 GB iii) 6400 GB
D) i) 3200 GB ii) 3200 GB iii) 6400 GB

RAID Summary

Level	Performance	Protection	Capacity (for N disks)
0	Best (parallel read/write)	Poor (lose all on 1 failure)	N
1	Good (write slower as 2x)	Good (have drive mirror)	N / 2
5	Good (must write parity block)	Good (one drive can fail)	N - 1
6	Good (must write multiple parity blocks)	Better (can have as many drives fail as dedicated to parity)	N – X (where X is # of parity drives such as 2)

File Interfaces

Besides the physical characteristics of the media and device, how the data is allocated on the media affects performance (***file organization***).

The physical device is controlled by the operating system. The operating system provides one or more interfaces to accessing the device.

Block-Level Interface

A ***block-level interface*** allows a program to read and write a chunk of memory called a ***block*** (or ***page***) from the device.

The page size is determined by the operating system. A page may be a multiple of the physical device's block or sector size.

The OS maintains a mapping from logical page numbers (starting at 0) to physical sectors/blocks on the device.

Block-Level Interface Operations

The block level operations at the OS level include:

- read(n,p) – read block *n* on disk into memory page *p*
- write(n,p) – write memory page *p* to block *n* on disk
- allocate(k,n) – allocate space for *k* contiguous blocks on device as close to block *n* as possible and return first block
- free(k,n) – marks *k* contiguous blocks starting at *n* as unused

The OS must maintain information on which blocks on the device are used and which are free.

Byte-Level Interface

A ***byte-level interface*** allows a program to read and write individually addressable bytes from the device.

A device will only directly support a byte-level interface if it is byte-addressable. However, the OS may provide a file-level byte interface to a device even if it is only block addressable.

File-Level Interface

A ***file-level interface*** abstracts away the device addressable characteristics and provides a standard byte-level interface for files to programs running on the OS.

A file is treated as a sequence of bytes starting from 0. File level commands allow for randomly navigating in the file and reading/writing at any location at the byte level.

Since a device may not support such access, the OS is responsible for mapping the logical byte address space in a file to physical device sectors/blocks. The OS performs buffering to hide I/O latency costs.

- Although beneficial, this level of abstraction may cause poor performance for I/O intensive operations.

Databases and File Interfaces

A database optimizes performance using device characteristics, so the file interface provided on the device is critical.

General rules:

- The database system needs to know block boundaries if the device is block addressable. It should not use the OS file interface mapping bytes to blocks.
  - ⇨ Full block I/Os should be used. Transferring groups of blocks is ideal.
- If the device has different performance for random versus sequential I/O and reads/writes, it should exploit this knowledge.
- If placement of blocks on the device matters, the database should control this not the OS.
- The database needs to perform its own buffering separate from the OS. Cannot use the OS virtual memory!

Databases and File Interfaces (2)

Two options:

- 1) Use a RAW block level interface to the device and manage everything. Very powerful but also a lot of complexity.
- 2) Use the OS file-level interface for data. Not suitable in general as OS hides buffering and block boundaries.

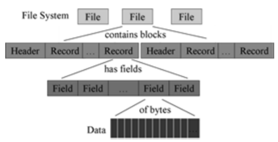
Compromise: Allocate data in OS files but treat files as raw disks. That is, do not read/write bytes but read/write to the file at the block level.

- The OS stills maps from logical blocks to physical blocks on the device and manages the device.
- BUT many performance issues with crossing block boundaries or reading/writing at the byte-level are avoided.
- Many systems make this compromise.

Representing Data in Databases Overview

A ***database*** is made up of one or more files.

- Each ***file*** contains one or more blocks.
- Each ***block*** has a header and contains one or more records.
- Each ***record*** contains one or more fields.
- Each ***field*** is a representation of a data item in a record.



COSC 404 - Dr. Ramon Lawrence

Representing Data in Memory

Consider an employee database where each employee record contains the following fields:

- name : string
- age : integer
- salary : double
- startDate : Date
- picture : BLOB

Each field is data that is represented as a sequence of bytes.

How would we store each field in memory or on disk?

COSC 404 - Dr. Ramon Lawrence

Representing Data in Memory Integers and Doubles

Integers are represented in two's complement format. The amount of space used depends on the machine architecture.

- e.g. `byte`, `short`, `int`, `long`

Double values are stored using a ***mantissa*** and an ***exponent***:

- Represent numbers in scientific format: $N = m * 2^e$
  - ⇨ *m* - mantissa, *e* - exponent, 2 - radix
  - ⇨ Note that converting from base 10 to base 2 is not always precise, since real numbers cannot be represented precisely in a fixed number of bits.
- The most common standard is **IEEE 754 Format**:
  - ⇨ 32 bit float - 1-bit sign; 8-bit exponent; 23-bit mantissa
  - ⇨ 64 bit double - 1-bit sign; 11-bit exponent; 52-bit mantissa

COSC 404 - Dr. Ramon Lawrence

Representing Data in Memory Doubles Example

The salary $56,455.01 stored as 4 consecutive bytes is:

- Hexadecimal value is: 475C8703 Stored value is: 56455.012

0 10001110 10111001000011100000011

sign bit exponent mantissa

- Divided into bytes looks like this:

Memory Address	F001	F002	F003	F004
	01000111	01011100	10000111	00000011

COSC 404 - Dr. Ramon Lawrence

Representing Data in Memory Strings and Characters

A ***character*** is represented by mapping the character symbol to a particular number.

- **ASCII** - maps characters/symbols to a number from 0 to 255.
- **UNICODE** - maps characters to a two-byte number (0 to 32,767) which allows for the encoding of larger alphabets.

A ***string*** is a sequence of characters allocated in consecutive memory bytes. A pointer indicates the location of the first byte.

- ***Null-terminated string*** - last byte value of 0 indicates end
- ***Byte-length string*** - length of string in bytes is specified (usually in the first few bytes before string starts).
- ***Fixed-length string*** - always the same size.

COSC 404 - Dr. Ramon Lawrence

Representing Data in Memory Dates

A ***date*** value can be represented in multiple ways:

- Integer representation - number of days past since a given date
  - ⇨ Example: # days since Jan 1, 1900
- String representation - represent a date's components (year, month, day) as individual characters of a string
  - ⇨ Example: YYYYMMDD or YYYYDDD
  - ⇨ Please do not reinvent Y2K by using YYMMDD!!

A ***time*** value can also be represented in similar ways:

- Integer representation - number of seconds since a given time
  - ⇨ Example: # of seconds since midnight
- String representation - hours, minutes, seconds, fractions
  - ⇨ Example: HHMMSSFF

Representing Data in Memory
BLOBs and Large Objects

A ***BLOB (Binary Large Object)*** type is represented as a sequence of consecutive bytes with the size of the object stored in the first few bytes.

All variable length types and objects will store a size as the first few bytes of the object.

Fixed length objects do not require a size, but may require a type identifier.

Storing Records in Memory

Now that we can allocate space for each field in memory, we must determine a way of allocating an entire record.

A ***record*** consists of one or more fields grouped together.

- Each tuple of a relation in the relational model is a record.

Two main types of records:

- ***Variable-length records*** - the size of the record varies.
- ***Fixed-length records*** - all records have the same size.

Separating Fields of a Record

The fields of a record can be separated in multiple ways:

- 1) ***No separator*** - store length of each field, so do not need a separate separator (fixed length field).
  - Simple but wastes space within a field.
- 2) ***Length indicator*** - store a length indicator at the start of the record (for the entire record) and a size in front of each field.
  - Wastes space for each length field and need to know length beforehand.
- 3) ***Use offsets*** – at start of record store offset to each field
- 4) ***Use delimiters*** - separate fields with delimiters such as a comma (comma-separated files).
  - Must make sure that delimiter character is not a valid character for field.
- 5) ***Use keywords*** - self-describing field names before field value (XML and JSON).
  - Wastes space by using field names.

Schemas

A ***schema*** is a description of the record layout.

A schema typically contains the following information:

- names and number of fields
- size and type of each field
- field ordering in record
- description or meaning of each field

Schemas
Fixed versus Variable Formats

If every record has the same fields with the same types, the schema defines a ***fixed record format***.

- Relational schemas generally define a fixed format structure.

It is also possible to have no schema (or a limited schema) such that not all records have the same fields or organization.

- Since each record may have its own format, the record data itself must be ***self-describing*** to indicate its contents.
- XML and JSON documents are considered self-describing with variable schemas (***variable record formats***).

Schemas
Fixed Format Example

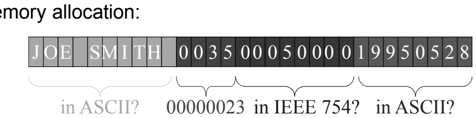
Employee record is a fixed relational schema format:

Field Name	Type	Size in Bytes
name	char(10)	10
age	integer	4
salary	double	8
startDate	Date	8 (YYYYMMDD)

Example record:

- Joe Smith, 35, $50,000, 1995/05/28

Memory allocation:



Schemas
Fixed Format with Variable fields

It is possible to have a fixed format (schema), yet have variable sized records.

- In the Employee example, the picture field is a BLOB which will vary in size depending on the type and quality of the image.

It is not efficient to allocate a set memory size for large objects, so the fixed record stores a pointer to the object and the size of the object which have fixed sizes.

The object itself is stored in a separate file or location from the rest of the records.

Variable Formats
XML and JSON

XML:

```
<employees>
<employee>
   <name>Joe Smith</name> <age>35</age>
   <salary>50000</salary> <hired>1995/05/28</hired>
</employee>
<employee>
   <name>CEO</name><age>55</age><hired>1994/06/23</hired>
</employee>
</employees>
```

JSON:

```
{ "employees": [ { "name":"Joe Smith", "age":35,
                  "salary":50000, "hired":"1995/05/28"},
                 { "name":"CEO", "age":55,
                  "hired":"1994/06/23"} ] }
```


Variable Format Discussion

Variable record formats are useful when:

- The data does not have a regular structure in most cases.
- The data values are sparse in the records.
- There are repeating fields in the records.
- The data evolves quickly so schema evolution is challenging.

Disadvantages of variable formats:

- Waste space by repeating schema information for every record.
- Allocating variable-sized records efficiently is challenging.
- Query processing is more difficult and less efficient when the structure of the data varies.

Format and Size Question

Question: JSON and XML are best described as:

A) fixed format, fixed size

B) fixed format, variable size

C) variable format, fixed size

D) variable format, variable size

Relational Format and Size Question

Question: A relational table uses a VARCHAR field for a person's name. It can be best described as:

A) fixed format, fixed size

B) fixed format, variable size

C) variable format, fixed size

D) variable format, variable size

Fixed vs. Variable Formats Discussion

There are also many variations that have properties of both fixed and variable format records:

- Can have a record type code at the beginning of each record to denote what fixed schema it belongs to.
  - ⇨ Allows the advantage of fixed schemas with the ability to define and store multiple record types per file.
- Define custom record headers within the data that is only used once.
  - ⇨ Do not need separate schema information, and do not repeat the schema information for every record.
- It is also possible to have a record with a fixed portion and a variable portion. The fixed portion is always present, while the variable portion lists only the fields that the record contains.

Fixed versus Variable Formats Discussion (2)

We have seen fixed length/fixed format records, and variable length/variable format records.

1) Do fixed format and variable length records make sense?

Yes, you can have a fixed format schema where certain types have differing sizes. BLOBs are one example.

2) Do variable format and fixed length records make sense?

Surprisingly, Yes. Allocate a fixed size record then put as many fields with different sizes as you want and pad the rest.

|320587 | Joe Smith | SC | 95 | 3 | ←Padding→
|184923 | Kathy Li | EN | 92 | 3 | ←Padding→
| 249793 | Albert Chan | SC | 94 | 3 |←Padding→

Research Question CHAR versus VARCHAR

Question: We can represent a person's name in MySQL using either CHAR(50) or VARCHAR(50). Assume that the person's name is 'Joe'. How much space is actually used?

A) CHAR = 3 ; VARCHAR = 3

B) CHAR = 50 ; VARCHAR = 3

C) CHAR = 50 ; VARCHAR = 4

D) CHAR = 50 ; VARCHAR = 50

Storing Records in Blocks

Now that we know how to represent entire records, we must determine how to store sets of records in blocks.

There are several issues related to storing records in blocks:

- 1) ***Separation*** - how do we separate adjacent records?
- 2) ***Spanning*** - can a record cross a block boundary?
- 3) ***Clustering*** - can a block store multiple record types?
- 4) ***Splitting*** - are records allocated in multiple blocks?
- 5) ***Ordering*** - are the records sorted in any way?
- 6) ***Addressing*** - how do we reference a given record?

Storing Records in Blocks Separation

If multiple records are allocated per block, we need to know when one record ends and another begins.

Record ***separation*** is easy if the records are a fixed size because we can calculate the end of the record from its start.

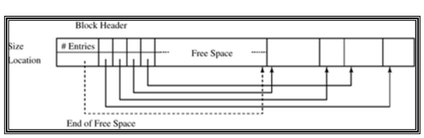
Variable length records can be separated by:

- 1) Using a special separator marker in the block.
- 2) Storing the size of the record at the start of each record.
- 3) Store the length or offset of each record in the block header.

Variable Length Records Separation and Addressing

A ***block header*** contains the number of records, the location and size of each record, and a pointer to block free space.

Records can be moved around within a block to keep them contiguous with no empty space between them and the header is updated accordingly.



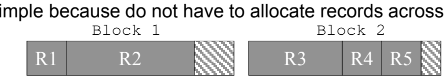
Storing Records in Blocks Spanning

If records do not exactly fit in a block, we have two choices:

- 1) Waste the space at the end of each block.
- 2) Start a record at the end of a block and continue on the next.

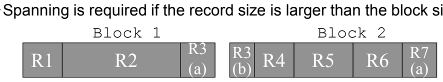
Choice #1 is the ***unspanned*** option.

- Simple because do not have to allocate records across blocks.



Choice #2 is the ***spanned*** option.

- Each piece must have a pointer to its other part.
  - ⇨ Spanning is required if the record size is larger than the block size.



Storing Records in Blocks Spanning Example

If the block size is 4096 bytes, the record size is 2050 bytes, and we have 1,000,000 records:

- How many blocks are needed for spanned/unspanned records?
- What is the block (space) utilization in both cases?

Answer:

- Unspanned
  - put one record per block implies 1,000,000 blocks
  - each block is only 2050/4096 * 100% = 50% full (utilization = 50%)
- Spanned
  - all blocks are completely full except the last one
  - # of blocks required = 1,000,000 * 2050 / 4096 = 500,049 blocks
  - utilization is almost 100%

Storing Records in Blocks Clustering

Clustering is allocating records of different types together on the same block (or same file) because they are frequently accessed together.

Example:

- Consider creating a block where a department record is allocated together with all employees in the department:

Block 1

DPT1	EMP1	EMP2	DEPT2	EMP3	EMP4

Storing Records in Blocks Clustering (2)

If the database commonly processes queries such as:

```
select * from employee, department
where employee.deptId = department.Id
```

then the clustering is beneficial because the information about the employee and department are adjacent in the same block.

However, for queries such as:

```
select * from employee

select * from department
```

clustering is harmful because the system must read in more blocks, as each block read contains information that is not needed to answer the query.

Storing Records in Blocks Split Records

A ***split record*** is a record where portions of the record are allocated on multiple blocks for reasons other than spanning.

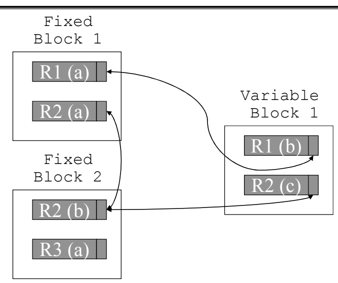
- Record splitting may be used with or without spanning.

Typically, hybrid records are allocated as split records:

- The ***fixed portion*** of the record is allocated on one block (with other fixed record portions).
- The ***variable portion*** of the record is allocated on another block (with other variable record portions).

Splitting a record is done for efficiency and simplifying allocation. The fixed portion of a record is easier to allocate and optimize for access than the variable portion.

Storing Records in Blocks Split Records with Spanning Example



Storing Records in Blocks Ordering Records

Ordering (or sequencing) records is when the records in a file (block) are sorted based on the value of one or more fields.

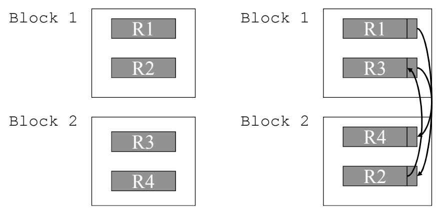
Sorting records allows some query operations to be performed faster including searching for keys and performing joins.

Records can either be:

- 1) ***physically ordered*** - the records are allocated in blocks in sorted order.
- 2) ***logically ordered*** - the records are not physical sorted, but each record contains a pointer to the next record in the sorted order.

Storing Records in Blocks
Ordering Records Example

Physical ordering **Logical Ordering**



What are the tradeoffs between the two approaches?
What are the tradeoffs of any ordering versus unordered?

Storing Records in Blocks
Addressing Records

Addressing records is a method for defining a unique value or address to reference a particular record.

Records can either be:

- 1) ***physically addressed*** - a record has a physical address based on the device where it is stored.
  - A physical disk address may use a sector # or a physical address range exposed by the device.
- 2) ***logically addressed*** - a record that is logically addressed has a key value or some other identifier that can be used to lookup its physical address in a table.
  - Logical addresses are indirect addresses because they provide a mechanism for looking up the actual physical addresses. They do not provide a method for locating the record directly on the device.
  - E.g. OS provides logical block to physical sector mapping for files.

Storing Records in Blocks
Addressing Records Tradeoff

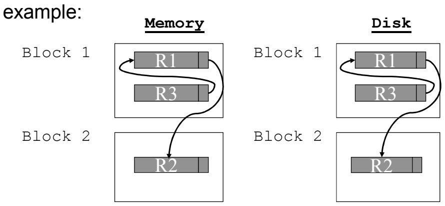
There is a tradeoff between physical and logical addressing:

- Physical addresses have better ***performance*** because the record can be accessed directly (no lookup cost).
- Logical addresses provide more ***flexibility*** because records can be moved on the physical device and only the mapping table needs to be updated.
  - The actual records or fields that use the logical address do not have to be changed.
  - Easier to move, update, and change records with logical addresses.

Pointer Swizzling

When transferring blocks between the disk and memory, we must be careful when handling pointers in the blocks.

For example:



Pointer swizzling is the process for converting disk pointers to memory pointers and vice versa when blocks move between memory and disk.

Operations on Files

Once data has been stored to a file consisting of blocks of records, the database system will perform operations such as update and delete to the stored records.

How records are allocated and addressed affects the performance for update and delete operations.

Operations on Files
Record Deletion

When a record is deleted from a block, we have several options:

- 1) Reclaim deleted space
  - Move another record to the location or compress file.
- 2) Mark deleted space as available for future use

Tradeoffs:

- Reclaiming space guarantees smaller files, but may be expensive especially if the file is ordered.
- Marking space as deleted wastes space and introduces complexities in maintaining a record of the free space available.

Operations on Files
Issues with Record Deletion

We must also be careful on how to handle references to a record that has been deleted.

- If we re-use the space by storing another record in the same location, how do we know that the correct record is returned or indicate the record has been deleted?

Solutions:

- 1) Track down and update all references to the record.
- 2) Leave a "tombstone" marker at the original address indicating record deletion and not overwrite that space.
  - Tombstone is in the block for physical addressing, in the lookup table for logical addressing.
- 3) Allocate a unique record id to every record and every pointer or reference to a record must indicate the record id desired.
  - Compare record id of pointer to record id of record at address to verify correct record is returned.

Research Question
PostgreSQL `VACUUM`

Question: What does the `VACUUM` command do in PostgreSQL?

A) Cleans up your dirty house for you

B) Deletes records from a given table

C) Reclaims space used by records marked as deleted

D) Removes tables no longer used

Operations on Files
Record Insertion

Inserting a record into a file is simple if the file is not ordered.

- The record is ***appended*** to the end of the file.

If the file is physically ordered, then all records must be shifted down to perform insert.

- Extremely costly operation!

Inserting into a logically ordered file is simpler because the record can be inserted anywhere there is free space and linked appropriately.

- However, a logically ordered file should be periodically re-organized to ensure that records with similar key values are in nearby blocks.

Memory and Buffer Management

Memory management involves utilizing buffers, cache, and various levels of memory in the memory hierarchy to achieve the best performance.

- A database system seeks to minimize the number of block transfers between the disk and memory.

A ***buffer*** is a portion of main memory available to store copies of disk blocks.

A ***buffer manager*** is a subsystem responsible for allocating buffer space in main memory.

Buffer Manager Operations

All read and write operations in the database go through the buffer manager. It performs the following operations:

- **read block *B*** – if block *B* is currently in buffer, return pointer to it, otherwise allocate space in buffer and read block from disk.
- **write block *B*** – update block *B* in buffer with new data.
- **pin block *B*** – request that *B* cannot be flushed from buffer
- **unpin block *B*** – remove pin on block *B*
- **output block *B*** – save block *B* to disk (can either be requested or done by buffer manager to save space)

Key challenge: How to decide which block to remove from the buffer if space needs to be found for a new block?

Buffer Management
Replacement Strategy

A ***buffer replacement strategy*** determine which block should be removed from the buffer when space is required.

- Note: When a block is removed from the buffer, it must be written to disk if it was modified. and replaced with a new block.

Some common strategies:

- Random replacement
- Least recently used (LRU)
- Most recently used (MRU)

Buffer Replacement Strategies and Database Performance

Operating systems typically use least recently used for buffer replacement with the idea that the past pattern of block references is a good predictor of future references.

However, database queries have well-defined access patterns (such as sequential scans), and a database system can use the information to better predict future references.

- LRU can be a bad strategy for certain access patterns involving repeated scans of data!

Buffer manager can use statistical information regarding the probability that a request will reference a particular relation.

- E.g., The schema is frequently accessed, so it makes sense to keep schema blocks in the buffer.

Research Question MySQL Buffer Management

Question: What buffer replacement policy does MySQL InnoDB use?

A) LRU

B) MRU

C) 2Q

Column Storage

The previous discussion on storage formats assumed records were allocated on blocks. For large data warehouses, it is more efficient to allocate data at the column level.

Each file represents all the data for a column. A file entry contains the column value and a record id. Records are rebuilt by combining columns using the record id.

The column format reduces the amount of data retrieved from disk (as most queries do not need all columns) and allows for better compression.

Research Question PostgreSQL Column Layout

Question: Does PostgreSQL support column layout?

A) Yes

B) No

Issues in Disk Organizations

There are many ways to organize information on a disk.

- There is no one correct way.

The "best" disk organization will be determined by a variety of factors such as: ***flexibility***, ***complexity***, ***space utilization***, and ***performance***.

Performance measures to evaluate a given strategy include:

- space utilization
- expected times to search for a record given a key, search for the next record, insert/append/delete/update records, reorganize the file, read the entire file.

Key terms:

- ***Storage structure*** is a particular organization of data.
- ***Access mechanism*** is an algorithm for manipulating the data in a storage structure.

Summary

Storage and Organization

`Hardware`	hard drives, RAID (formulas) sequential/random access
`Fields`	representing types in memory
`Records`	variable/fixed format/length schemas
`Blocks`	separation, spanning, splitting, clustering, ordering, addressing
`Files`	insert, delete operations on various organizations
`Memory`	buffer management pointer swizzling
`Database`	disk organization choices

Major Objectives

The "One Things":

- Perform device calculations such as computing transfer times.
- Explain the differences between fixed and variable schemas.
- List and briefly explain the six record placement issues in blocks.

Major Theme:

- There is no single correct organization of data on disk. The "best" disk organization will be determined by a variety of factors such as: ***flexibility***, ***complexity***, ***space utilization***, and ***performance***.

Objectives

- Compare/contrast volatile versus non-volatile memory.
- Compare/contrast random access versus sequential access.
- Perform conversion from bytes to KB to MB to GB.
- Define terms from hard drives: arm assembly, arm, read-write head, platter, spindle, track, cylinder, sector, disk controller
- Calculate disk performance measures - capacity, access time (seek,latency,transfer time), data transfer rate, mean time to failure.
- Explain difference between sectors (physical) & blocks (logical).
- Perform hard drive and device calculations.
- List the benefits of RAID and common RAID levels.
- Explain issues in representing floating point numbers.

Objectives (2)

- List different ways for representing strings in memory.
- List different ways for representing date/times in memory.
- Explain the difference between fixed and variable length records.
- Compare/contrast the ways of separating fields in a record.
- Define and explain the role of schemas.
- Compare/contrast variable and fixed formats.
- List and briefly explain the six record placement issues in blocks.
- Explain the tradeoffs for physical/logical ordering and addressing.
- List the methods for handling record insertion/deletion in a file.
- List some buffer replacement strategies.
- Explain the need for pointer swizzling.
- Define storage structure and access mechanism.