Outline

- Storage hierarchy
- Disks
- RAID
- File Organization
- Etc....

Storage Hierarchy

- Main memory
  - 10s or 100s of ns; volatile
  - Pretty cheap and dropping: 1GByte << $100
  - Main memory databases feasible now-a-days
- Flash memory (EEPROM)
  - Limited number of write/erase cycles
  - Non-volatile, slower than main memory (especially writes)
  - Examples?
- Question
  - *How does what we discuss next change if we use flash memory only?*
  - *Key issue: Random access as cheap as sequential access*
Accessing Data

- Accessing a sector
  - Time to seek to the track (seek time)
    - average 4 to 10ms
  - + Waiting for the sector to get under the head (rotational latency)
    - average 4 to 11ms
  - + Time to transfer the data (transfer time)
    - very low
  - About 10ms per access
    - So if randomly accessed blocks, can only do 100 block transfers
    - 100 x 512bytes = 50 KB/s

- Data transfer rates
  - Rate at which data can be transferred (w/o any seeks)
  - 30-50MB/s to up to 200MB/s (Compare to above)
    - Seeks are bad!
Disk Controller

- Interface between the disk and the CPU
- Accepts the commands
- *checksums* to verify correctness
- Remaps bad sectors

---

Optimizing block accesses

- Typically sectors too small
- Block: A contiguous sequence of sectors
  - 4k to 16k
  - All data transfers done in units of blocks
- Scheduling of block access requests?
  - Considerations: *performance* and *fairness*
  - *Elevator algorithm*
**Solid State Drives**

- Essentially flash that emulates hard disk interfaces
- No seeks → Much better random reads performance
- Writes are slower, the number of writes at the same location limited
  - Must write an entire block at a time
- About a factor of 2x more expensive right now (2023)

- Lead to perhaps the most radical hardware configuration change in a while

**Flash Storage**

- NOR flash vs NAND flash
- NAND flash
  - used widely for storage, cheaper than NOR flash
  - requires page-at-a-time read (page: 512 bytes to 4 KB)
    - 20 to 100 microseconds for a page read
    - Not much difference between sequential and random read
  - Page can only be written once
    - Must be erased to allow rewrite
- Solid state disks
  - Use standard block-oriented disk interfaces, but store data on multiple flash storage devices internally
  - Transfer rate of up to 500 MB/sec using SATA, and up to 3 GB/sec using NVMe PCIe
Flash Storage (Cont.)

- Erase happens in units of **erase block**
  - Takes 2 to 5 milliseconds
  - Erase block typically 256 KB to 1 MB (128 to 256 pages)
- **Remapping** of logical page addresses to physical page addresses avoids waiting for erase
- Flash translation table tracks mapping
  - also stored in a label field of flash page
  - remapping carried out by flash translation layer
- After 100,000 to 1,000,000 erases, erase block becomes unreliable and cannot be used
  - wear leveling

SSD Performance Metrics

- Random reads/writes per second
  - Typical 4 KB reads: 10,000 reads per second (10,000 IOPS)
  - Typical 4KB writes: 40,000 IOPS
  - SSDs support parallel reads
    - Typical 4KB reads:
      - 100,000 IOPS with 32 requests in parallel (QD-32) on SATA
      - 350,000 IOPS with QD-32 on NVMe PCIe
    - Typical 4KB writes:
      - 100,000 IOPS with QD-32, even higher on some models
- Data transfer rate for sequential reads/writes
  - 400 MB/sec for SATA3, 2 to 3 GB/sec using NVMe PCIe
- Hybrid disks: combine small amount of flash cache with larger magnetic disk
Storage Class Memory

- 3D-XPoint memory technology pioneered by Intel
- Available as Intel Optane
  - SSD interface shipped from 2017
    - Allows lower latency than flash SSDs
  - Non-volatile memory interface announced in 2018
    - Supports direct access to words, at speeds comparable to main-memory speeds
- Expensive

Outline

- Storage hierarchy
- Disks
- RAID
- File Organization
- Etc....
Reliability

- Mean time to/between failure (MTTF/MTBF):
  - 57 to 136 years
- Consider:
  - 1000 new disks
  - 1,200,000 hours of MTTF each
  - Components in series:
    \[
    \text{mtbf}(c_1; c_2) = \frac{1}{\text{mtbf}(c_1) + \frac{1}{\text{mtbf}(c_2)}} = \frac{\text{mtbf}(c_1) \times \text{mtbf}(c_2)}{\text{mtbf}(c_1) + \text{mtbf}(c_2)}
    \]
    - On average, one will fail 1200 hours = 50 days!

RAID

- Redundant array of independent disks
- Goal:
  - Disks are very cheap
  - Failures are very costly
  - Use “extra” disks to ensure reliability
    - If one disk goes down, the data still survives
    - Also allows faster access to data
- Many raid “levels”
  - Different reliability and performance properties
Redundant Array Independent Disks

Fast!

Redundant!

Weird!

RAID Level 5

- Distributed parity “blocks” instead of bits
- Normal operation:
  - “Read” directly from single disk.
    - Load distributed across all 5 disks
  - “Write”: Need to read and update the parity block
    - To update 9 to 9’
      - read 9 and P2
      - compute P2’ = P2 $\oplus$ 9 $\oplus$ 9’
      - write 9’ and P2’

(f) RAID 5: block-interleaved distributed parity
RAID Level 5

- Failure operation (disk 3 has failed)
  - “Read block 0”: Read it directly from disk 2
  - “Read block 1” (which is on disk 3)
    - Read P0, 0, 2, 3 and compute \(1 = P0 \oplus 0 \oplus 2 \oplus 3\)
  - “Write”:
    - To update 9 to 9’
      - read 9 and P2
        - Oh... P2 is on disk 3
        - So no need to read or update it
      - Write 9’

Choosing a RAID level

- RAID 0 striping fastest, but no fault tolerance
- Main choice between RAID 1 and RAID 5
- Level 1 better write performance than level 5
  - Level 5: 2 block reads and 2 block writes to write a single block
  - Level 1: only requires 2 block writes
  - Level 1 preferred for high update environments such as log disks
- Level 5 lower storage cost
  - Usable storage for Level 1 only 50% of raw disk capacity
  - Level 5 is preferred for applications with low update rate, and large amounts of data
Query Processing/Storage

- Given a query, decide how to “execute” it
- Specify sequence of pages to be brought in memory
- Operate upon the tuples to produce results

Buffer Manager

- When the QP wants a block, it asks the “buffer manager”
  - The block must be in memory to operate upon
- Buffer manager:
  - If block already in memory: return a pointer to it
  - If not:
    - Evict a current page
      - Either write it to temporary storage,
      - or write it back to its original location,
      - or just throw it away (if it was read from disk, and not modified)
    - and make a request to the storage subsystem to fetch it

- Bringing pages from disk to memory
- Managing the limited memory

- Storage hierarchy
- How are relations mapped to files?
- How are tuples mapped to disk blocks?
Buffer Manager

- Similar to *virtual memory manager*
- Buffer replacement policies
  - What page to evict?
  - LRU: Least Recently Used
    - Throw out the page that was not used in the longest time
  - Clock?
    - An efficient implementation of LRU

Choice of frame dictated by replacement policy.
Buffer Manager Must Allow

- **Pinning** a block
  - Not allowed to evict
- **Force-output (force-write)**
  - Force the contents of a block to be written to disk
- **Order the writes**
  - This block must be written to disk before this block

Critical for fault tolerant guarantees
- Otherwise database has no control over what is on disk

Outline

- Storage hierarchy
- Disks
- RAID
- Buffer Manager
- File Organization
- Etc…..
File Organization

- How are the relations mapped to the disk blocks?
  - Use a standard file system?
    - High-end systems have their own OS/file systems
    - OS interferes more than helps in many cases
  - Mapping of relations to file?
    - One-to-one?
    - Advantages in storing multiple relations clustered together
  - A file is essentially a collection of disk blocks
    - How are the tuples mapped to the disk blocks?
    - How are they stored within each block

File Organization

- Goals:
  - Allow insertion/deletions of tuples/records
  - Fetch a particular record (specified by record id)
  - Find all tuples that match a condition (say SSN = 123)?
- Simplest case
  - Each relation is mapped to a file
  - A file contains a sequence of records
  - Each record corresponds to a logical tuple
- Next:
  - How are tuples/records stored within a block?
Fixed Length Records

- \( n \) = number of bytes per record
- Store record \( i \) at position:
  - \( n \times (i - 1) \)
- Records may cross blocks
  - Not desirable
  - Stagger so that that doesn’t happen
- Inserting a tuple?
  - Depends on the policy used
  - One option: Simply append at the end of the file. Problems?

- Deletions?
  - Option 1: Rearrange
  - Option 2: Keep a free list and use for next insert

The above assumes records not ordered.

Fixed Length Records

- Deleting: using “free lists”

<table>
<thead>
<tr>
<th>Record</th>
<th>Name</th>
<th>Department</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10101</td>
<td>Srinivasan</td>
<td>65000</td>
</tr>
<tr>
<td>1</td>
<td>15151</td>
<td>Mozart</td>
<td>40000</td>
</tr>
<tr>
<td>2</td>
<td>22222</td>
<td>Einstein</td>
<td>95000</td>
</tr>
<tr>
<td>3</td>
<td>33456</td>
<td>Gold</td>
<td>87000</td>
</tr>
<tr>
<td>4</td>
<td>58583</td>
<td>Califrier</td>
<td>62000</td>
</tr>
<tr>
<td>5</td>
<td>76543</td>
<td>Singh</td>
<td>80000</td>
</tr>
<tr>
<td>6</td>
<td>76766</td>
<td>Crick</td>
<td>72000</td>
</tr>
<tr>
<td>7</td>
<td>83982</td>
<td>Brandt</td>
<td>90000</td>
</tr>
<tr>
<td>8</td>
<td>98345</td>
<td>Kim</td>
<td>80000</td>
</tr>
</tbody>
</table>
Variable-length Records

Slotted page structure

- **Indirection:**
  - The records may move inside the page, but the outside world is oblivious to it
  - Why?
    - The headers are used as an indirection mechanism
    - *Record ID 1000 is the 5th entry in the page number X*

Sequential File Organization

- What if I want to find a particular record by value?
  - *Account info for SSN = 123*
  - sequential == sorted
  - Binary search.
    - Takes \( \text{ceiling}(\log_2(n)) \) number of disk accesses
      - Random accesses
    - Too much
      - \( n = 1,000,000,000 - \log_2(n) = 30 \)
      - Recall each random access approx 10 ms
      - 300 ms to find just one account information
      - < 4 requests satisfied per second
Outline

- Storage hierarchy
- Disks
- RAID
- Buffer Manager
- File Organization
- Indexes
- Etc…

File Organization

- Which block of a file should a record go to?
  - Anywhere?
    - How to search for “SSN = 123”? 
    - Called “heap” organization
  - Sorted by SSN?
    - Called “sequential” organization
    - Keeping it sorted would be painful
    - How would you search?
  - Based on a “hash” key
    - Called “hashing” organization
    - Store the record with SSN = x in the block number x%1000
    - Why?
Sequential File Organization

- Keep sorted by some search key
- Insertion
  - Find the block in which the tuple should be
  - If there is free space, insert it
  - Otherwise, must create overflow pages
- Deletions
  - Delete and keep the free space
  - Databases tend to be insert heavy, so free space gets used fast
- Can become fragmented
  - Must reorganize once in a while

Index

- A data structure for efficient search through large databases
- Two key ideas:
  - The records are mapped to the disk blocks in specific ways
    - Sorted, or hash-based
    - Auxiliary data structures are maintained that allow quick search
- Think library index/catalogue
- “Search key”:
  - Attribute or set of attributes used to look up records
  - E.g. SSN for a persons table
  - Can be different from candidate or primary keys
- Types of indexes
  - Ordered indexes
  - Hash-based indexes
  - B+-trees
Ordered Indexes

- **Primary** (clustered) index
  - The relation is sorted on the *search key* of the index
- **Secondary** (nonclustered) index
  - It is not
  - Can have only one primary index on a relation

### Primary Sparse Index (vs dense)

- Every key doesn’t have to appear in the index
- Allows for very small indexes
  - Better chance of fitting in memory
  - Tradeoffs?
    - Some amount of in-memory search
    - Must access the relation file even if the record is not present