University of Glasgow | School of Computing Science

Introduction to Data Science and Systems

Zaiqiao Meng
Zaiqiao.Meng@glasgow.ac.uk

Lecture 8 - Storage

A bit of admin

- Lecture
  - All slides **without** any codes.
- Moodle quiz
  - Enabled just before this lecture.
  - Accepting answers in the next **48 hours**.
  - You will have **two** attempts.
- Lab4
  - Enabled just before this lecture.
  - Not only coding, but also some calculations with pen and paper.
  - Address data science tasks using techniques that you have learned previously.

Resources

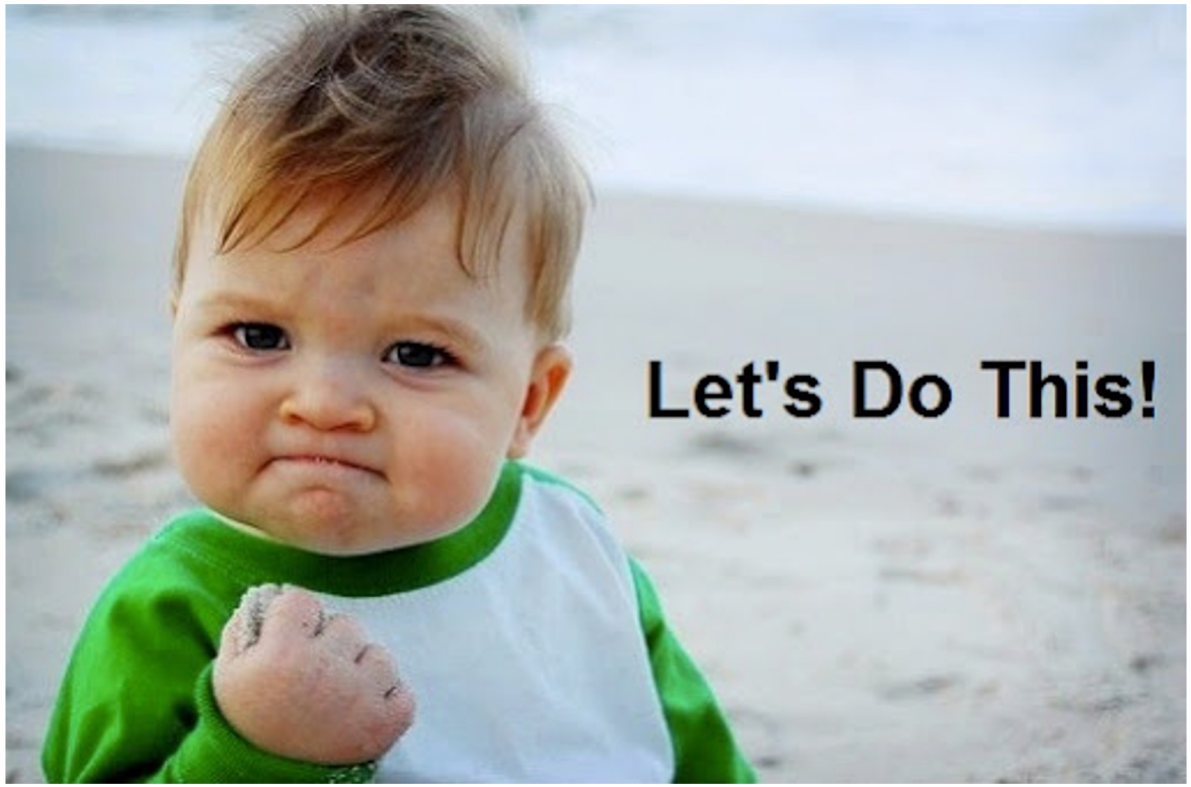
- Recommended Textbooks:
  - H. Garcia-Molina, J.D. Ullman and J. Widom. *"Database Systems: The Complete Book"*. Pearson Education Ltd, 2014.
    - Online access via UoG Library: http://tinyurl.com/ybsbthan
  - M. Kleppmann. Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems. O'Reilly, 2017.
    - Google around… ;-)
- Preliminary knowledge
  - Relational databases systems: most popular type of data-intensive system (MySQL, Oracle, etc).
  - SQL. E.g.

```
SELECT * FROM C, S WHERE C.Soil_type = S.Soil_type
```

    - It is ok if you don't know. https://www.w3schools.com/sql/

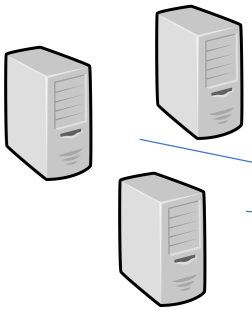
Outline

- Areas we'll touch upon:
  - Physical storage
  - Indexing
  - Query processing algorithms
  - Cost-based query optimisation


Let's Do This!

An Example of Basic Data System

Clients / users



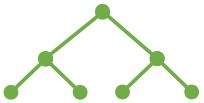
Queries

Data mgmt. system

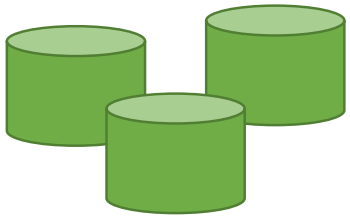
Logical dataset (e.g. table, graph)

First Name	Last Name	Address	City	Age
Mickey	Mouse	123 Fantasy Way	Anaheim	73
Bat	Man	321 Cavern Ave	Gotham	54
Wonder	Woman	987 Truth Way	Paradise	39
Donald	Duck	555 Quack Street	Mallard	65
Bugs	Bunny	567 Carrot Street	Rascal	58
Wiley	Coyote	999 Acme Way	Canyon	61
Cat	Woman	234 Purrfect Street	Hairball	32
Tweety	Bird	543	Itotitaw	28

Administrator



Physical storage (data structures)



What do you expect a *database system* to do?

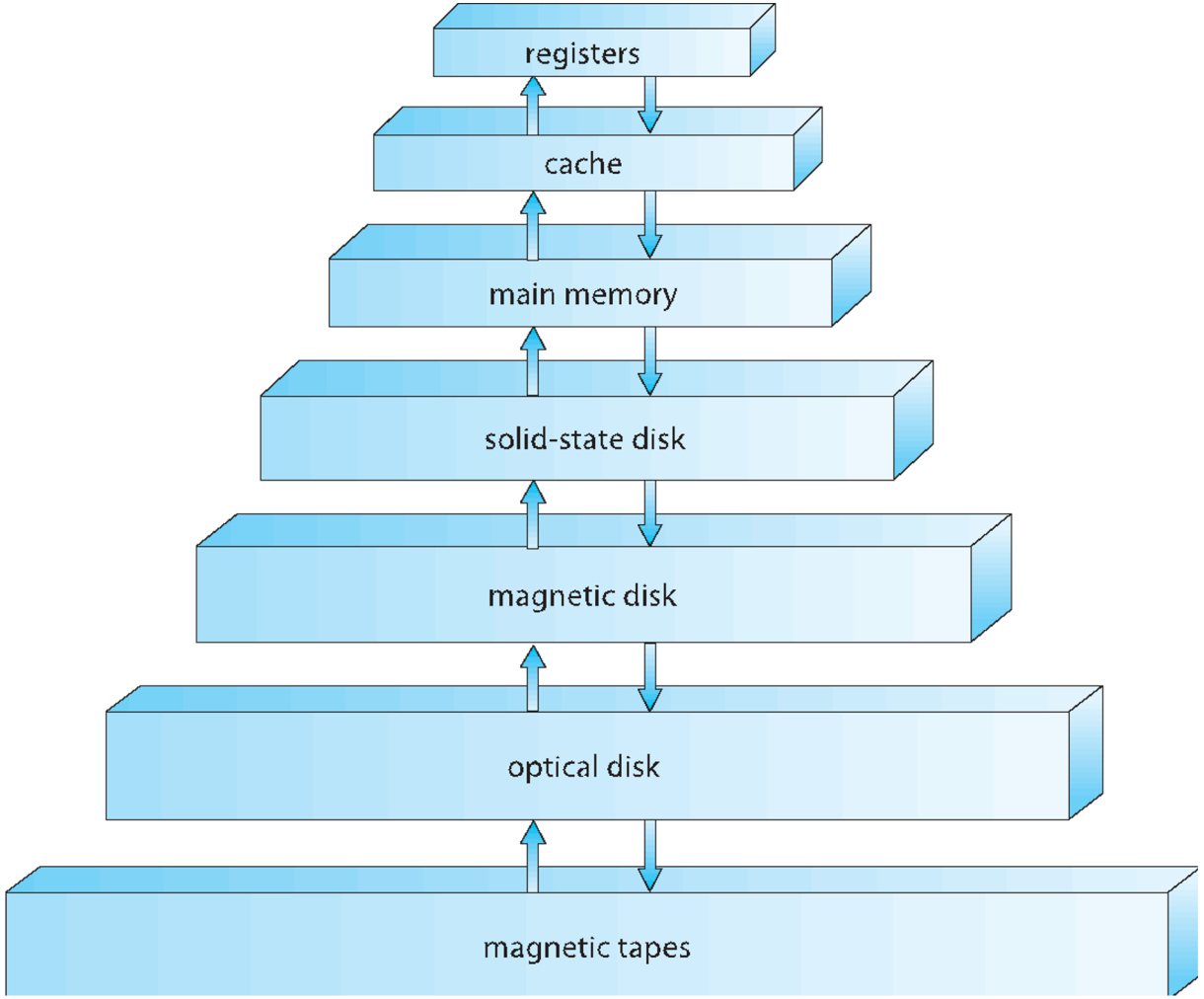
- Store data
  - Where?
    - Disk? Cloud? Ether? (*CAP theorem*)
  - How?
    - CSV files? Record files? Other?
  - For how long?
    - Stable storage? In-memory? (*durability*)
  - Guarantees?
    - Atomicity? Consistency? Isolation? (*ACID* vs *BASE*)
- Answer queries
  - Quickly…
    - Batch vs (near-)real-time
  - Accurately
    - Exact vs approximate results
  - Necessary components
    - Indexing, metadata, query processing, query optimisation

What is this?

What is this?


Stereo
Mono
Dolby
Nr.
1
BASE
chromdioxid
SM
60
Stereo
Mono
Dolby
Nr.
2
BASE
chromdioxid
SM
60

Storage-Device Hierarchy



Source: A. Silberschatz, "Operating System Concepts", 9th Ed., 2012.

Aside: System Latencies

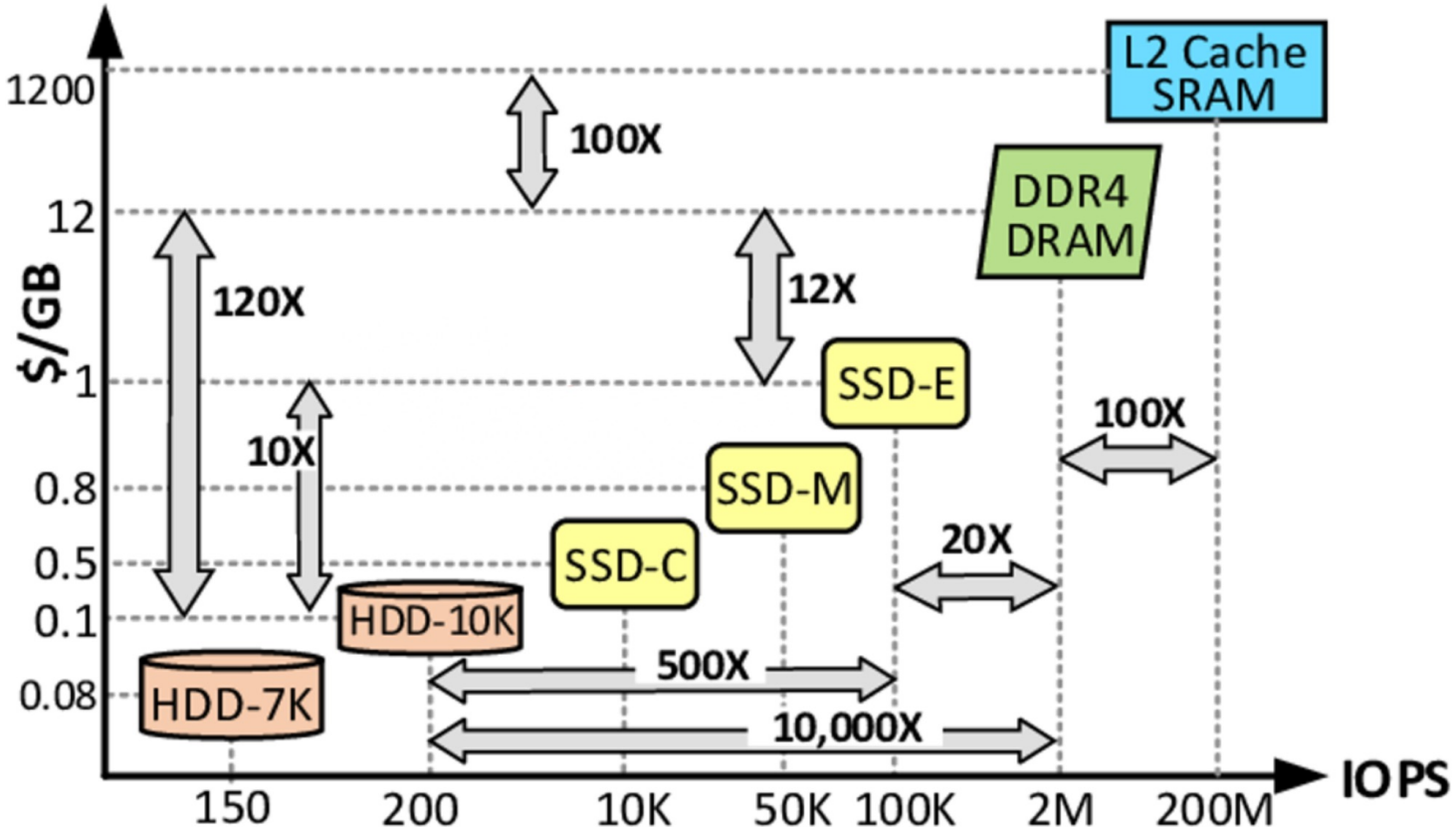
Event	Latency	Scaled
1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access (DRAM, from CPU)	120 ns	6 min
Solid-state disk I/O (flash memory)	50–150 µs	2–6 days
Rotational disk I/O	1–10 ms	1–12 months
Internet: San Francisco to New York	40 ms	4 years
Internet: San Francisco to United Kingdom	81 ms	8 years
Internet: San Francisco to Australia	183 ms	19 years
TCP packet retransmit	1–3 s	105–317 years
OS virtualization system reboot	4 s	423 years
SCSI command time-out	30 s	3 millennia
Hardware (HW) virtualization system reboot	40 s	4 millennia
Physical system reboot	5 m	32 millennia

@ B. Gregg, "Systems Performance: Enterprise and the Cloud", Prentice-Hall, 2013.

That would be awesome if we fit everything in **registers!**

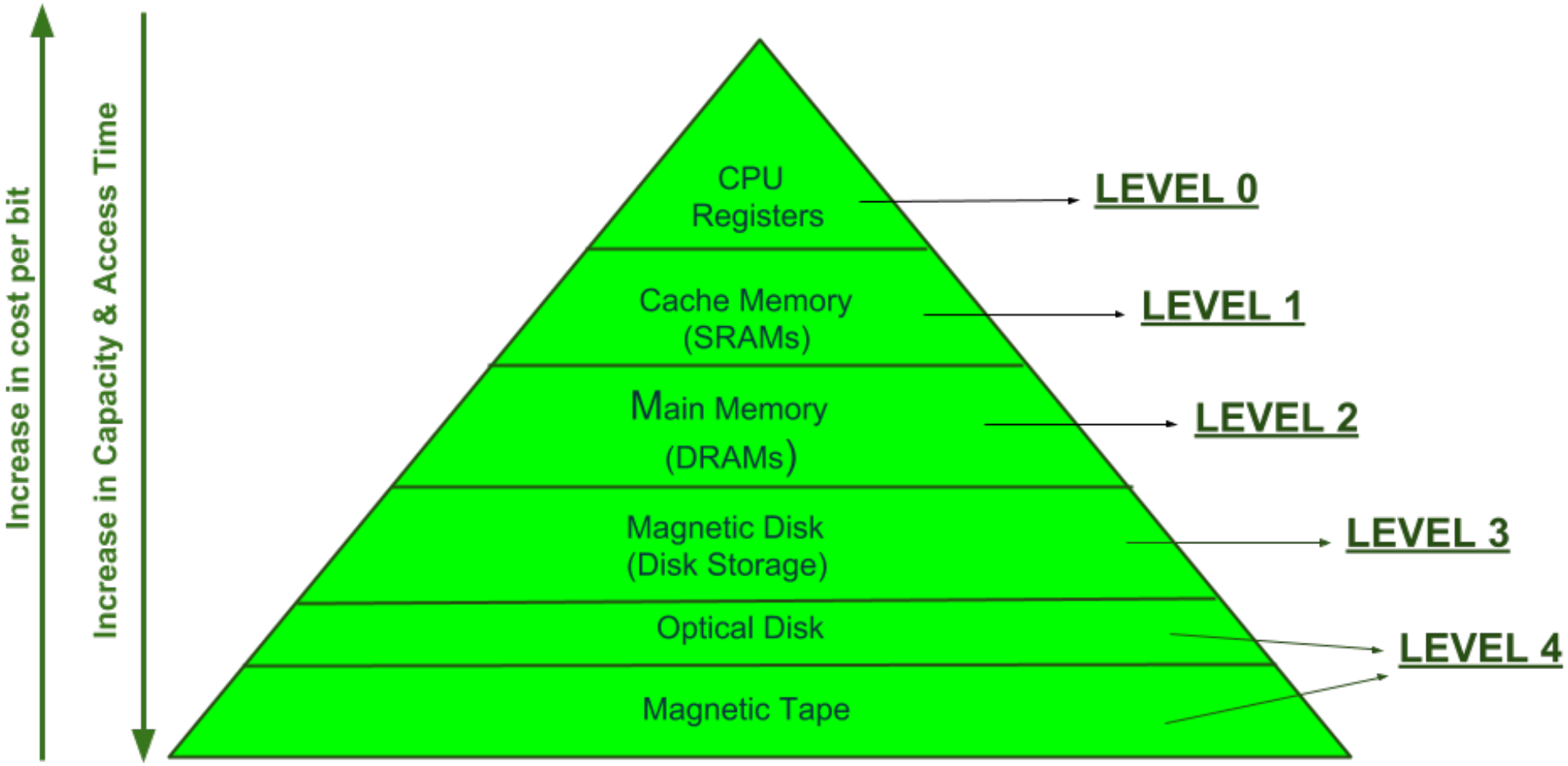
That would be awesome if we fit everything in **registers!**

However,



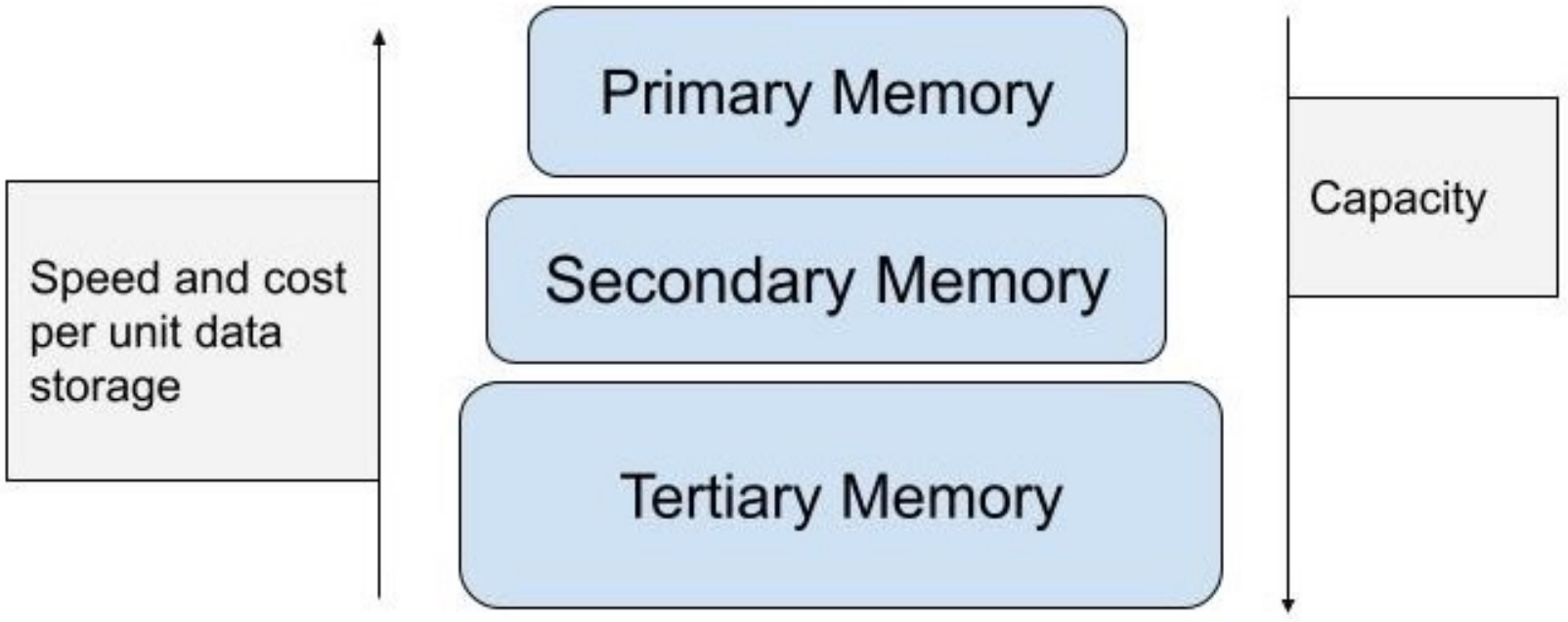
ETICA: Efficient Two-Level I/O Caching Architecture for Virtualized Platforms, TPDS 2021

Physical Storage Hierarchy



MEMORY HIERARCHY DESIGN

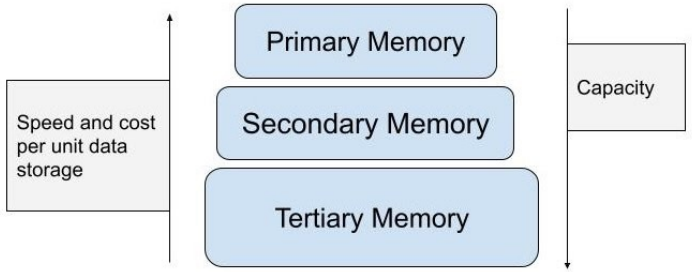
3-level hierarchy for data systems



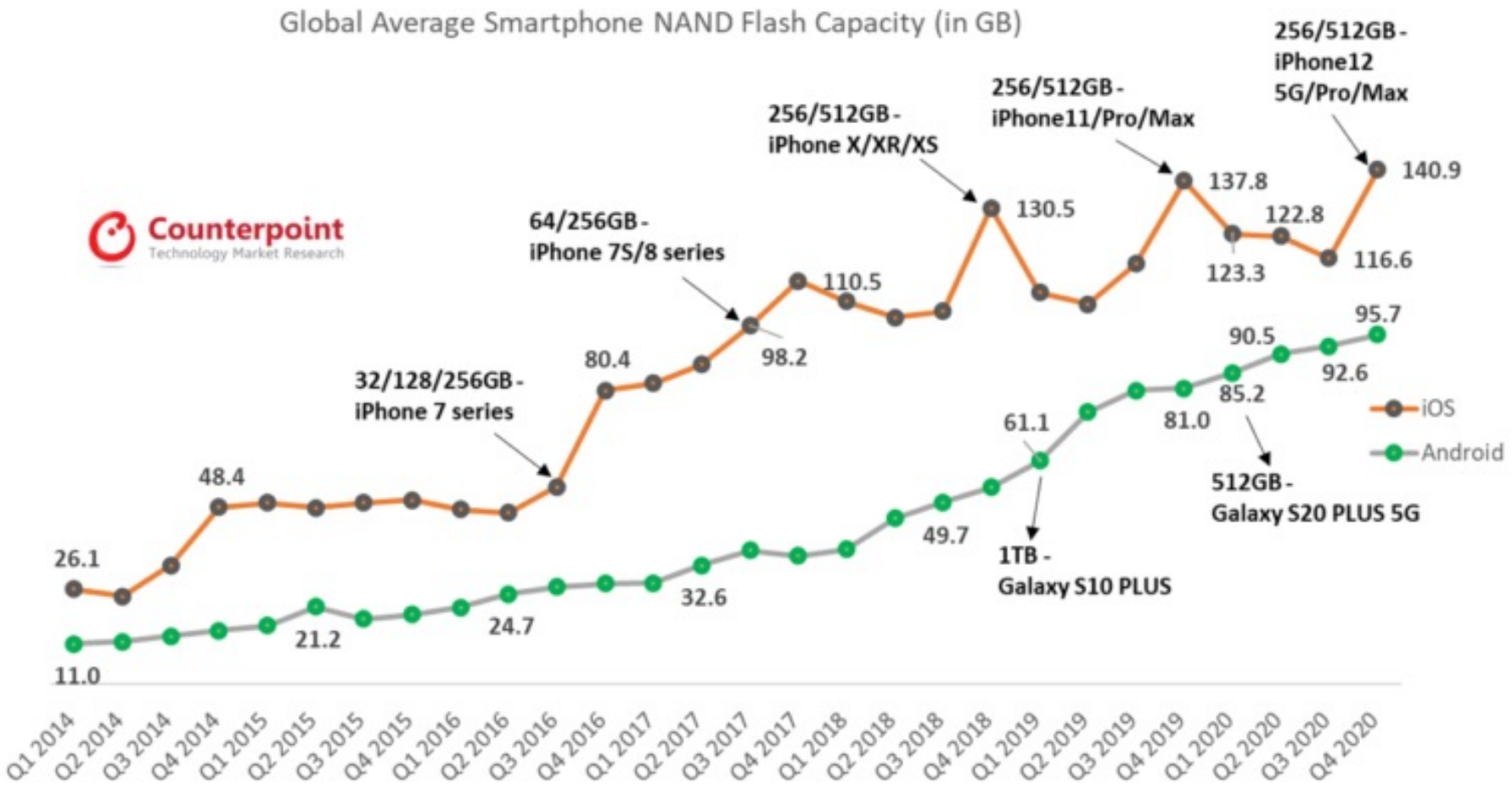
Physical Storage

- 3-level storage hierarchy
  - Primary storage
    - e.g., RAM: main memory, cache;
  - Secondary storage
    - e.g., hard-drive disks (HDD), solid-state disks (SSD);
  - Tertiary storage
    - e.g., optical drives.

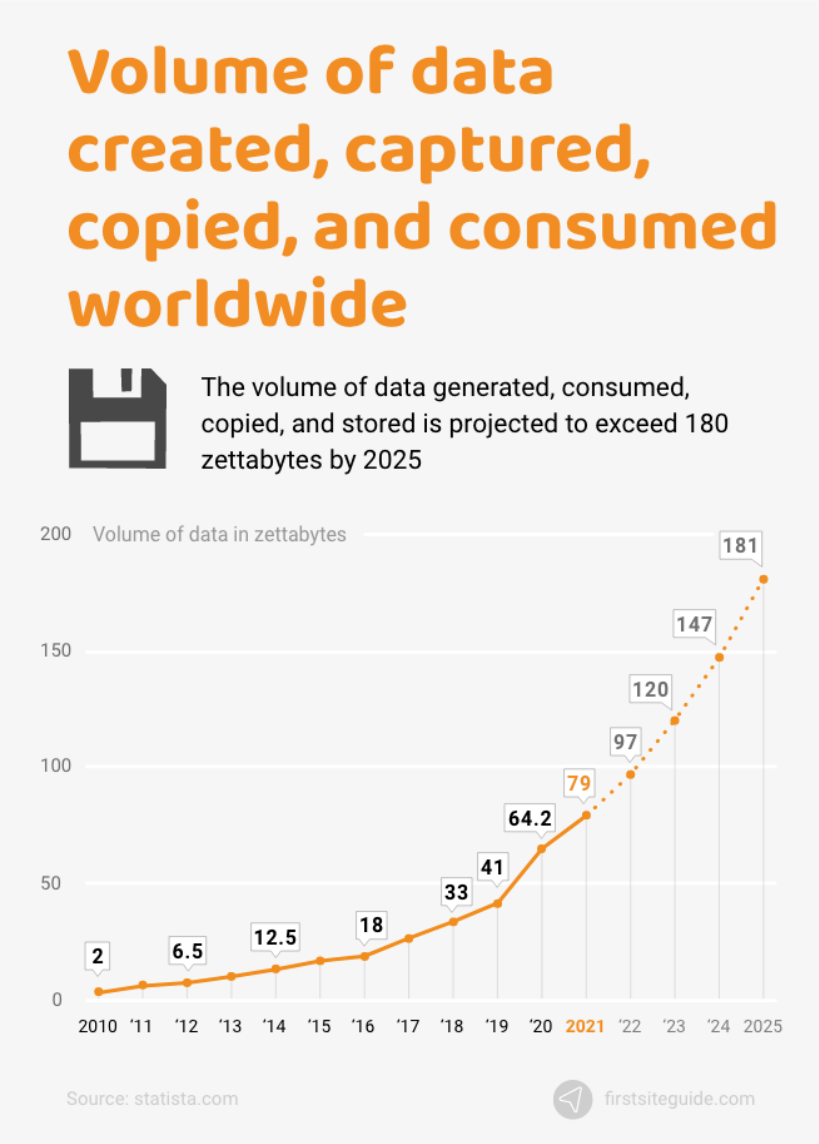
- As we go down the storage hierarchy
  - Storage capacity: increases
  - Access speed: decreases
  - Money-costs: decreases

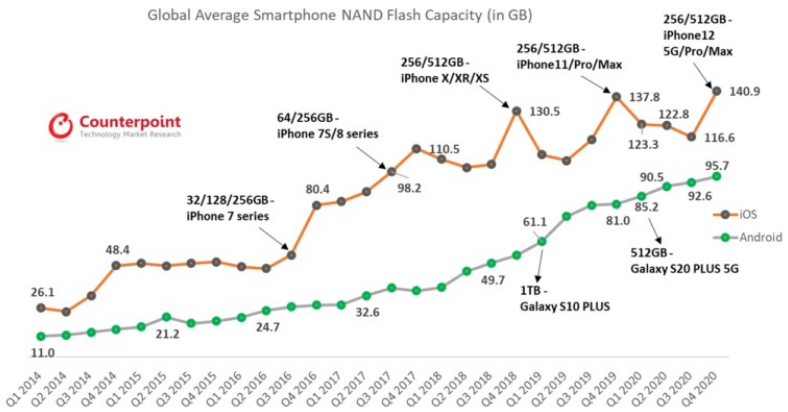


Size of RAM with Moore's law



Size of RAM vs Volume of Data

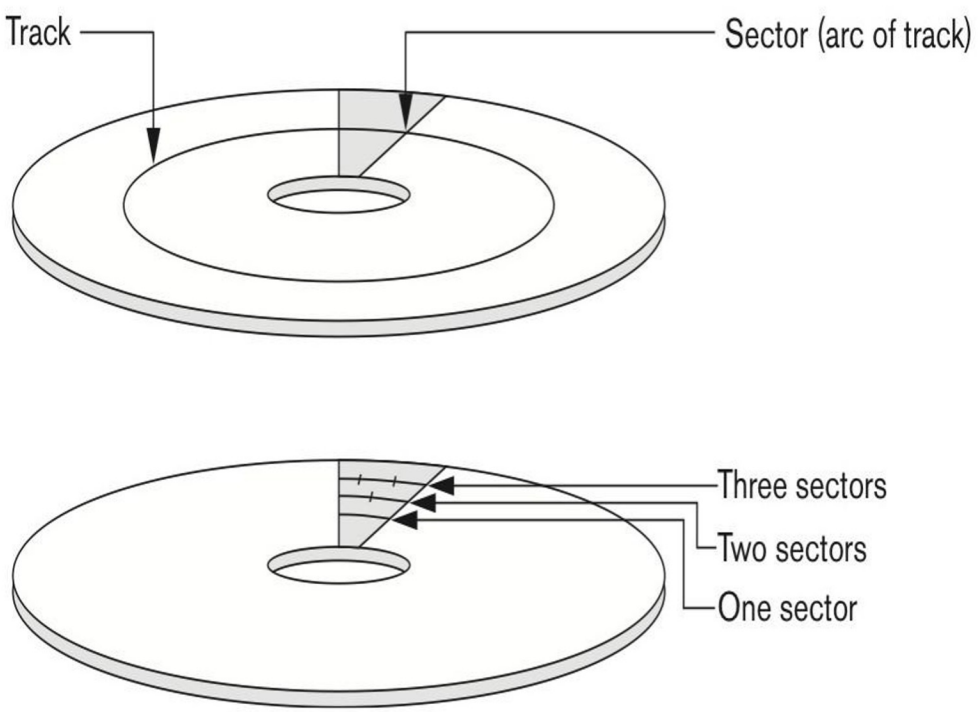




Physical Storage for a Database

- Fundamental Limitation: Database is too large to fit in RAM
  - By default, data lives on secondary storage
- But the CPU cannot directly access data not in RAM
  - Data must first be loaded into main memory from secondary storage…
  - … then fetched into registers to be processed
- Secondary storage access time becomes the bottleneck
  - Data access takes about 10ms-60ms, instead of 30-120ns
  - HDD 10,000x slower, SSD 20x slower.
- How can we alleviate this bottleneck?

Disk Anatomy & Data on Disk

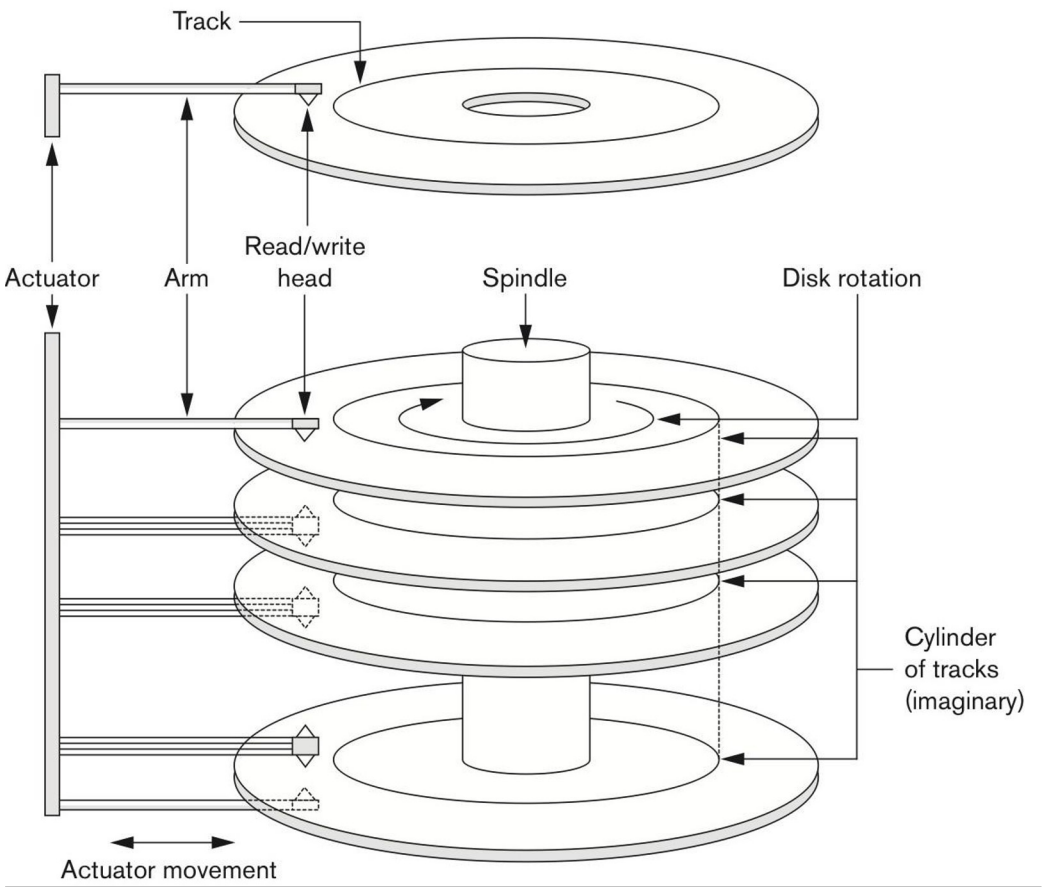


Recipe:

1. Take a (*magnetic*) disk
2. Dig *concentric* tracks
3. Split tracks into *circular* sectors
4. Take the DB data
5. Organise it in *files* of *records*
6. Organise each file in *blocks* (~1-10KBs)
7. Store each *block* in a set of sectors; each sector: ~512bytes - a few KBs
8. Done!

Note: DB is usually implemented on top of a File System; part of Operating System

Disk Anatomy & I/O Access



Read/Write Access (I/O)

- **Position** r/w heads:
  - *Seek delay*: ~3-6ms
- **Spin** until selected sector under the head:
  - *Rotation delay*: ~3-5ms
- **Transfer** data from track to disk buffer (extremely fast)
- **Transfer** data from disk buffer to RAM via bus (using DMA -- also fast)

Optimisation Problem: Spatially *organise* the file blocks in such an *order* to minimize the expected seek & rotation time

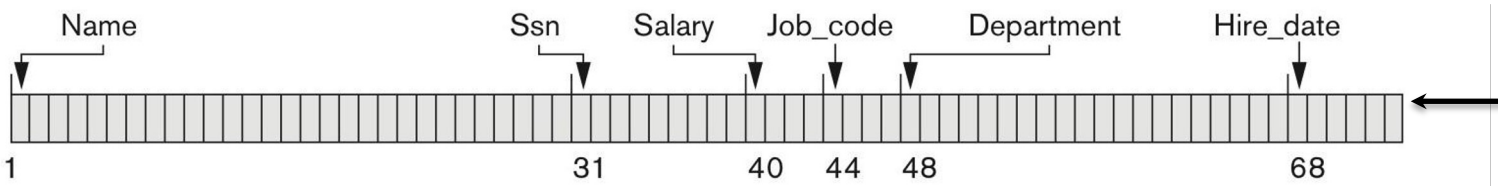
Organisation-based Optimisation

- Challenge: Organise tuples on the disk to minimizing I/O accesses

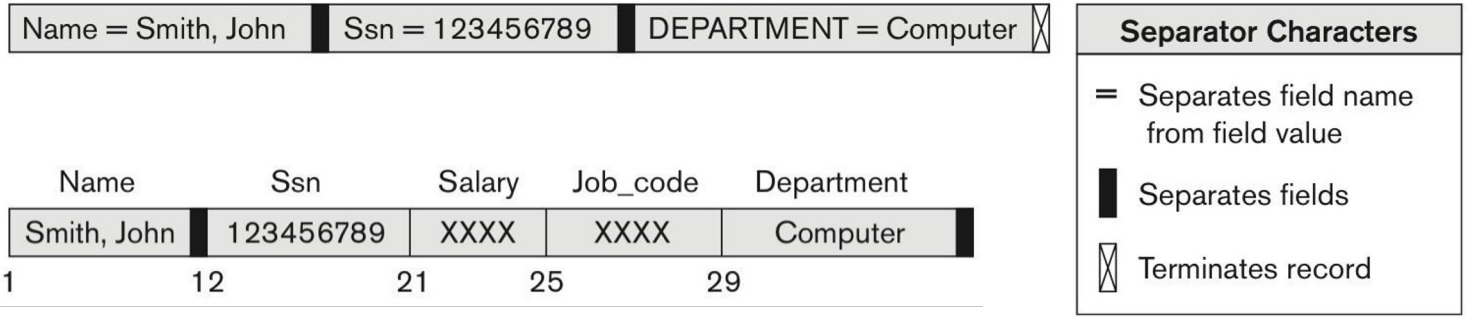
- Logical Representation
  - Represent a Tuple of a relation as a *record*
  - DB data is a set of records; each record is a set of attributes
  - Records are grouped together forming a *file*

- Records can be of:
  - Fixed length; i.e., the size is pre-determined by the designer
  - Variable length; i.e., the size of each attributes varies, e.g., some fields are optional with size >= 0 bytes

Fixed- vs variable-length records



- How to accommodate variable length records?
  - Option 1: "Magic" sequences as separators



  - Option 2: Prepend size of field, in fixed-length representation

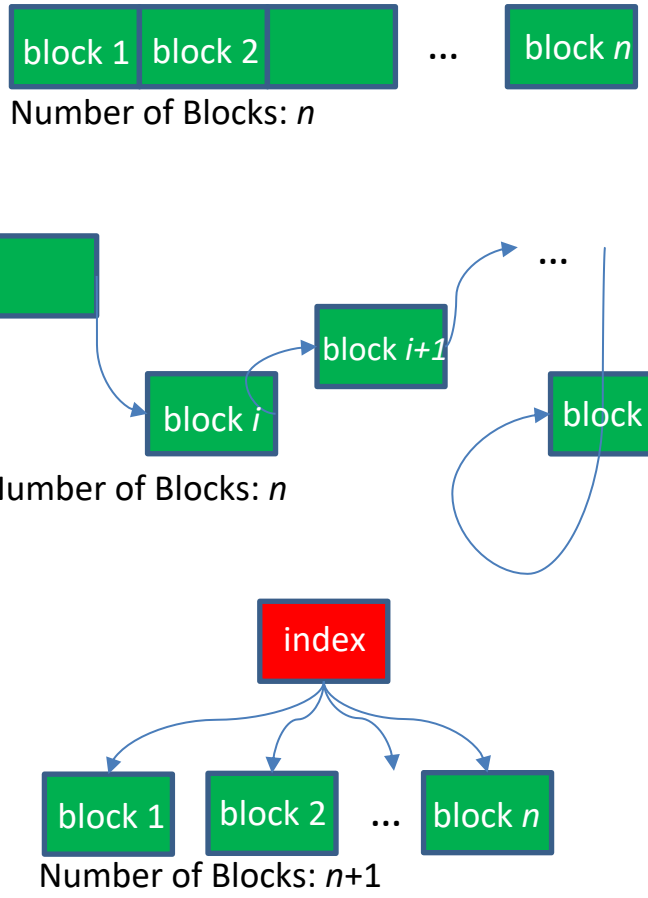
Field length (32-bit int)	Field data (Name = …)	Field length (32-bit int)	Field data (Ssn = …)	Field length (32-bit int)	Field data (Dept = …)

Blocking Factor

- Physical Representation
  - *Records* are stored in *blocks*
  - Blocks are of *fixed-length*, normally 512 bytes to 4096 bytes
- Consider a record of R bytes and a block of B bytes.
- **Blocking factor (bfr)**: The number of records stored in a block (i.e., records per block)
  - bfr = $\lfloor B/R \rfloor$ or floor(B/R)
  - e.g., bfr = 100 records per block means a block can store up to 100 records

Blocks to Files on Disk

- *File* is a set of *blocks*, and each block a set of *records*
- How do we allocate blocks to files?
  - **Contiguous (neighboring) allocation**: A file contains spatially consecutive blocks (a.k.a. self-navigation file)
  - **Linked allocation**: Each block i has a pointer to the logically next block i+1 anywhere on the disk -- i.e., a linked list of blocks; navigation information is stored in each block of the file
  - **Indexed allocation**: there exists a specific block (index block -- can be anywhere on the disk), which maintains a list of pointers pointing to the physical address of each block; navigation information stored in the index block



In-Class Example (placeholder)

- Context: The relation EMPLOYEE has r = 1103 tuples, each one corresponding to a fixed-length record. Each record has the fields:
  - NAME (30 bytes)
  - SSN (10 bytes)
  - ADDRESS (60 bytes)

- Given a block size of B = 512 bytes (defined by the OS):
  - Compute the blocking factor of the file that accommodates this relation

  - Compute the number of blocks in the file

File Structures for Database Data

- Challenge: Distribute records in a file to minimize I/O cost
- Options:
  - Heap Files, a.k.a. unordered files
    - Principle: new records are added at the end of the file, i.e., at the end of the last block (append)
  - Ordered Files, a.k.a. sequential files
    - Principle: records are kept sorted based on ordering field; if it is a key, it is called ordering key
  - Hash Files
    - Principle: a hashing function y = h(x) is applied to each record field x (hash field or hash key if it is a key)
    - The output y refers to the block id which contains record x → mapping a record to a block
- Given a file structure, we measure I/O cost for:
  - Retrieving a record x according to a searching field from disk to memory (retrieval/search cost)
  - Inserting/deleting/updating a record x from memory to the disk (update cost)
  - **Cost Function**: # of block accesses (read/write) to search/insert/delete/update record x

Heap File

- Inserting a new record is efficient:
  - Retrieve (load from disk to memory) the last block (addresses are kept at file header)
  - Insert the new record at the end of the block and write the block back to disk
  - Complexity: **O(1)** block access...
    - If it is full, then what?

- Retrieving a record is slow -- essentially, linear search through all the b file blocks
  - Retrieve each block from disk to memory
  - Search in-memory for the record
  - Repeat until EOF…
    - On average, we access ~ b/2 blocks
    - We access b blocks if the record is not in the file
  - Complexity: **O(b)** block accesses

- Deleting a record is slow:
  - Find and load the block containing the record
    - Remove data from the block and write the block back to disk → creates "holes" in blocks
  - Complexity: O(b) + O(1) block accesses
  - Alternatively, use deletion markers (aka tombstone records) instead (set a bit from 0 to 1)
    - Periodically, the storage space for the file is re-organised/compacted

Sequential File

- All the records are ordered by an ordering field, e.g., name, and are kept sorted at all times
  - Suitable for queries requiring sequential processing, queries on the ordering field or range queries over the ordering field
- Retrieval:
  - Retrieve a record using the ordering field: efficient
    - The right block can be found using binary search on the ordering field over a file with b blocks
    - Complexity is O(log2b)
  - Retrieve a record using a non-ordering field: inefficient
    - We do not exploit the record ordering --> equivalent to a heap file
    - Complexity is O(b) -- i.e., linear search
  - Range queries?
    - Efficient if on ordering field, otherwise inefficient
    - Methodology:
      - Using binary search, find the block which contains the record for the lower bound --> O(log2b)
      - Then, fetch contiguous blocks until the record containing the upper bound is located --> O(b)
    - Complexity: O(log2b) + O(b) block accesses
      = O(b)…

Sequential File

- Insertions:
  - Expensive!
    - Using binary search, locate the block where the record should be inserted, then move all subsequent records to make room for the new one
    - On average, half of the records must be moved to make room for the new record -- **very expensive**, especially for large files!
  - Alternative with chain pointers
    - If there is free space in the right block, insert the new record there, else insert the new record in an overflow block and use *chain pointers* -- pointer chain must be updated (akin to an on-disk sorted-linked list)
- Deletions:
  - Are expensive
    - Using binary search, locate the block where the record is to be deleted; binary search
    - Adopt deletion markers and update the pointer not to point to the deleted record
    - Periodically: reorganise/re-sort the file to restore the sequential order (invoke: external sorting… **expensive**)
- Updates:
  - On the ordering field value are costly
    - Record deleted from its old position & inserted into its new position in the file
  - On a non-ordering field value are efficient!
    - Complexity is $O(\log_2 b) + O(1)$ block accesses

Expected Block Accesses

Type of Organization	Access/Search Method	Average Time to Access a Specific Record
Heap (Unordered)	Sequential scan (Linear Search)	$b/2$
Ordered	Sequential scan	$b/2$
Ordered	Binary Search	$\log_2 b$

Rhetoric Question: *Why* the binary search over b blocks requires $\log_2 b$ accesses?

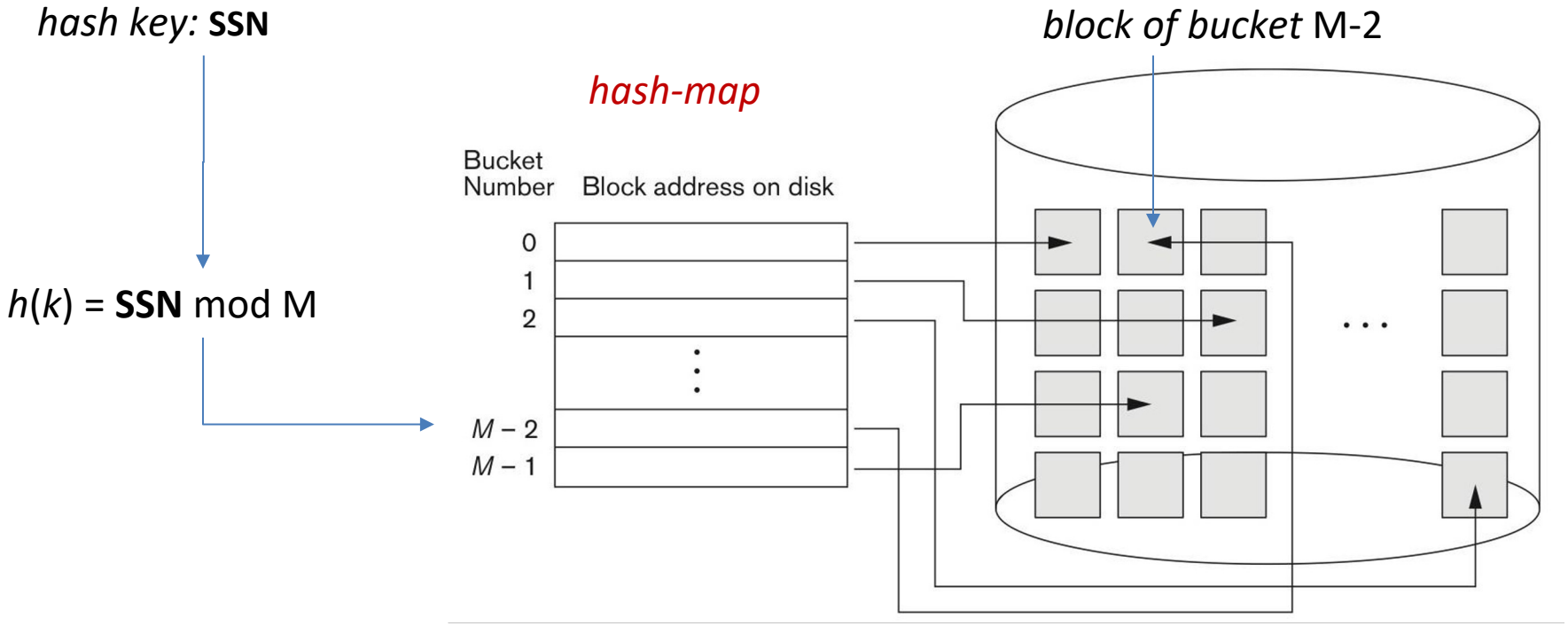
Answer: Logarithm of x indicates the *number of divisions* we need to divide x by 2 to reach 1, *e.g.,* $log_2(140) = 7.12$ steps to divide 140 by 2 to reach 1

We split the search space into two sub-spaces, and we repeat that…until finding the block!
What if we could split into 3, or more subspaces every time?
We then expedite the SQL retrieval process ☺️

Hash File

- What if most of our queries are selections using the equality predicate over a specific field *k*?
  - E.g., find the record *x* such that *x.k* = 23
- Idea: Use hashing
  - Partition (quantize) the records into M buckets: bucket 0, ..., bucket M-1 using a hash function y = h(k) with output y ∈ {0, 1, …, M-1}
    - Each bucket can have more than one blocks
  - h(k) has to uniformly distribute its input values over the M buckets
    - i.e., given a value k, each bucket will be chosen with equal probability 1/M
    - y = h(k) = k mod M
  - Mapping a record x to a bucket y = h(x.k) is called external hashing
    - Also called *indirect clustering*: group tuples together w.r.t. their hashed-values and not their hash-field values
  - Normally, collisions occur! i.e., two or more records are mapped to the same bucket!
    - Example: Let M = 3, h(k) = k mod 3 ∈ {0, 1, 2}, thus, we obtain three buckets
    - The records with k = 3, 11, 2, and 4 are stored in buckets 0, 2, 2, and 1, resp. → collision on bucket 2

External Hashing Retrieval



Retrieve a possible record by hashing key SSN:

1. Hash *k* and get the corresponding bucket, e.g., *h*(SSN) = M-2
2. Use the *file header* to translate it to the block address in disk
3. Fetch the block from the disk to memory
4. Linear search in memory to find the record *x* such that: *h*(SSN) = M-2

External Hashing Algorithm

- **Hash Search** (*k*: hash key)
  1. **Pre-Initialization: Load** the **meta-data** (hash) block with the *file header* from disk to memory, which contains the *hash-map* (*bucket*, *block-address*)
     - One block access
  2. **Compute** the bucket id $y = h(k)$
  3. **Access** the in-memory **hash-map** to get the block address related to bucket $y=h(k)$
  4. **Load** the block from the disk to memory
     - One block access
  5. **Search** in-memory the record *x* with hash $h(x.k)$
  6. **If** found **Then return** record *x* **Else return** not found

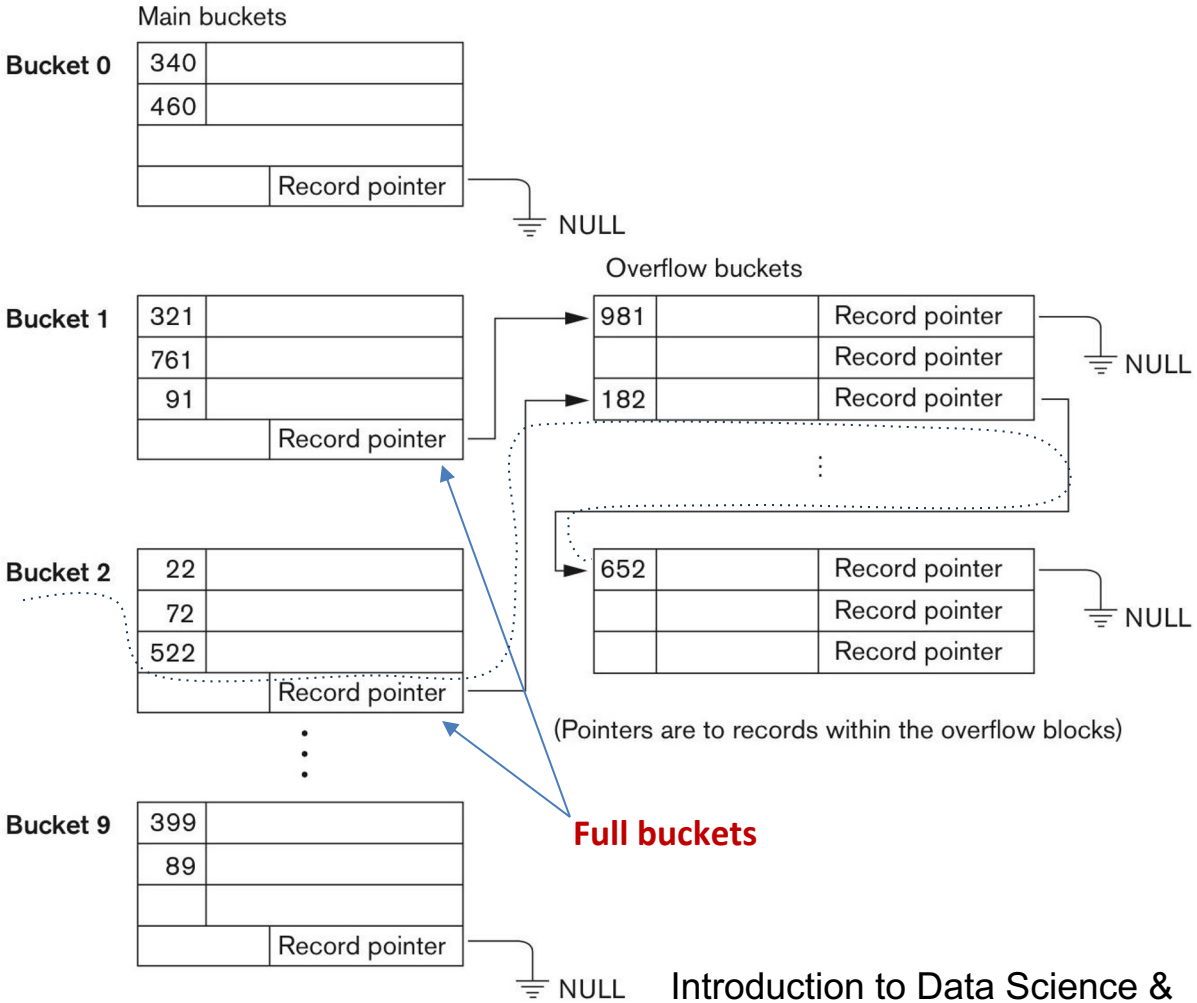
- **Complexity: O(1) block access**
  - i.e., *directly* get the block containing the record

External Hashing: Challenges

- Due to collisions, some buckets might be full
- Problem: How can we insert a new record hashed to a full bucket?
- Solution: Chain pointers to the rescue (again)

- Methodology:
  - Introduce overflow buckets to store records mapped to full buckets
  - At the end of each bucket, adopt a pointer to an overflow bucket; initially the pointer is NULL, thus no-overflow
  - If a bucket overflows, the pointer points to the corresponding overflow bucket
  - If more than one overflow buckets are needed then, at the end of each overflow bucket, we store another pointer to another overflow bucket…
- Search Complexity: O(1) + O(n) block accesses
  - n = number of overflow blocks

External Hashing Overflow



Overflowing records are found by:

- *following* pointers in their main bucket, first, and
- then *following* the pointers of records in overflow buckets

The hash keys **22, 72, 522, 182, 652** have hash value: **2**
(i.e., belong to bucket 2)

External Hashing

- Deletions:
  - Deleting a record x based on the hash field k is easy
    - If record x is in the main bucket then delete it from there immediately: O(1)
    - Note: We may need/want to move a record from the overflow bucket to the main bucket upon deletion
  - Deleting a record based on a non-hash field?
    - Requires scanning all records to identify those with that value… Expensive!
- Updates:
  - Updating the value of a non-hash field m ≠ k of a record x is easy
    (1) Locate record x in its main or overflow bucket(s)
    (2) Load block into memory, update it, and write it back
    - O(1) or O(1) + O(n) block accesses
  - Updating the value of the hash field k (to k') for a record x a bit more complicated
    - Deletion of old record in O(1)
    - Insertion to the new bucket referred to by h(x.k') in O(1) in the best case

In-Class Example

- Assume:
  - Number of buckets: M = 3
  - 1 block per bucket
  - bfr = 2 records/block

Name	Ssn	Salary	Job_code	Department
Smith, John	123456789	XXXX	XXXX	Computer

- SSN values: {1000; 4540; 4541; 4323; 1321; 1330}
  - Assign the Employees to Buckets w.r.t. SSN and y = SSN mod 3
    - 1000 mod 3 = 1; 4540 mod 3 = 1; 4541 mod 3 = 2;
      4323 mod 3 = 0; 1321 mod 3 = 1; 1330 mod 3 = 1
  - Calculate the expected number of block accesses for a random SQL query: SELECT * FROM EMPLOYEE WHERE SSN = k …in the worst case!
    - Bucket 0: 1 record (**1 block**); Bucket 1: 4 records (**2 blocks**); Bucket 2: 1 record (**1 block**)
    - **What is the access pattern?**
    - If each bucket is equi-probable to be selected, thus, queried with probability 1/3:
      0.33*1 + 0.33*(1 + 1) + 0.33*1 = 1.32 block accesses
    - If each SSN value is equi-probable to be selected:
      1/6 * 1 + 2/3*(1 + 1) + 1/6 * 1 = 2 block accesses
    - Linear scan (Heap File): 4 block accesses (worst case!)
    - Sequential File (SSN-ordered): $\log_2(4) = 2$ block accesses!

External Hashing (placeholder)

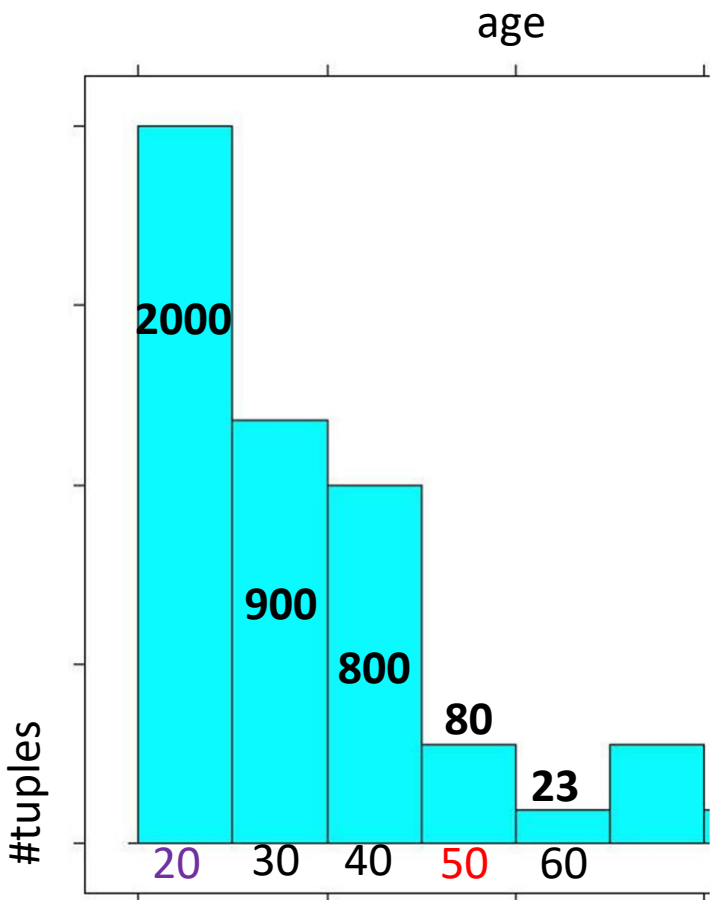
- Range queries in hash file:
  - Inefficient to execute
- Consider a relationship Employee with Age as a hash key attribute, and the query:
  - SELECT * FROM Employee WHERE Age BETWEEN 20 AND 50

- Methodology?

2 mins to think about it . . .

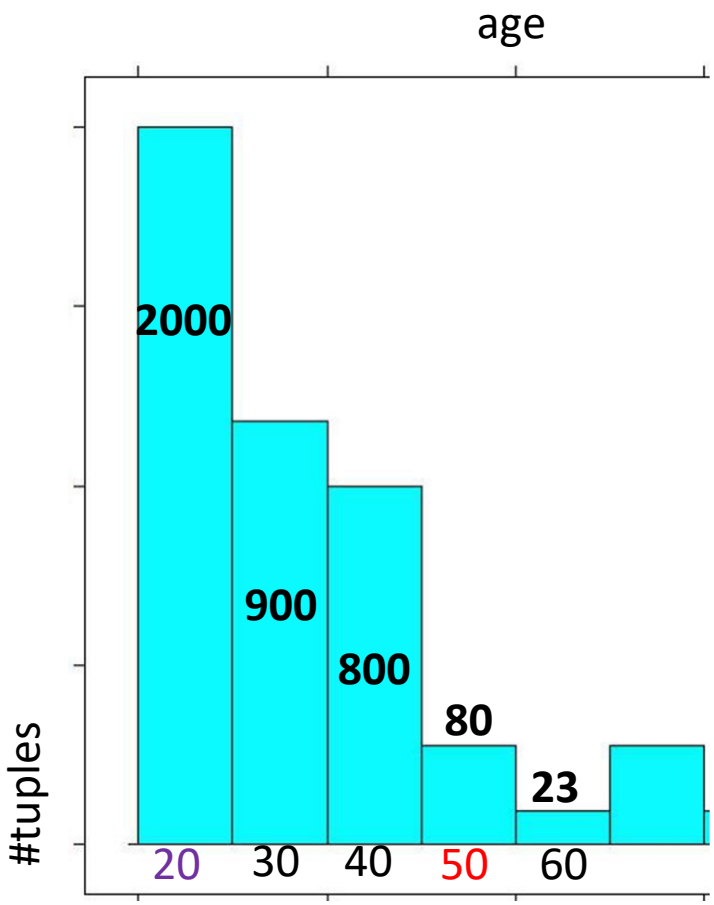
Predictable or Unpredictable? (placeholder)

- Note: The distribution of the values influences the expected cost
  - The expected cost is unpredictable!



Predictable or Unpredictable?

- Note: The distribution of the values influences the expected cost
  - The expected cost is unpredictable!
- Experiment 1: Retrieve the record x with x.age = 20
  - **O(1) + O(*m*) block accesses** including *m* overflow blocks ☹
- Experiment 2: Retrieve the record x with x.age = 60 (bfr>23)
  - **O(1) = 1 block access** ☺
- The distribution determines the load per bucket!
  - Hash h should be chosen based on the distribution and not arbitrarily!



External Hashing: Limitations

- We cannot efficiently scan sequentially the hash file block-by-block since it requires, first, sorting of the records;
  - Not suitable for SQL queries involving ORDER BY x.k ☹
- Supports only search with the equality predicate '='
  - Not suitable for SQL queries involving negation NOT or '< >' x.k ☹
- It is not applicable for range queries since logically continuous values are not stored in physically contiguous blocks ☹
  - …due to the hash function (uniform distribution of values)
- The number of buckets M is fixed, thus, there is a problem when the number of records in the file grows or shrinks ☹
  - i.e., it cannot hash dynamically changing files

Special Thanks

Special Thanks to Dr Nikos Ntarmos who is the original author of the slides.