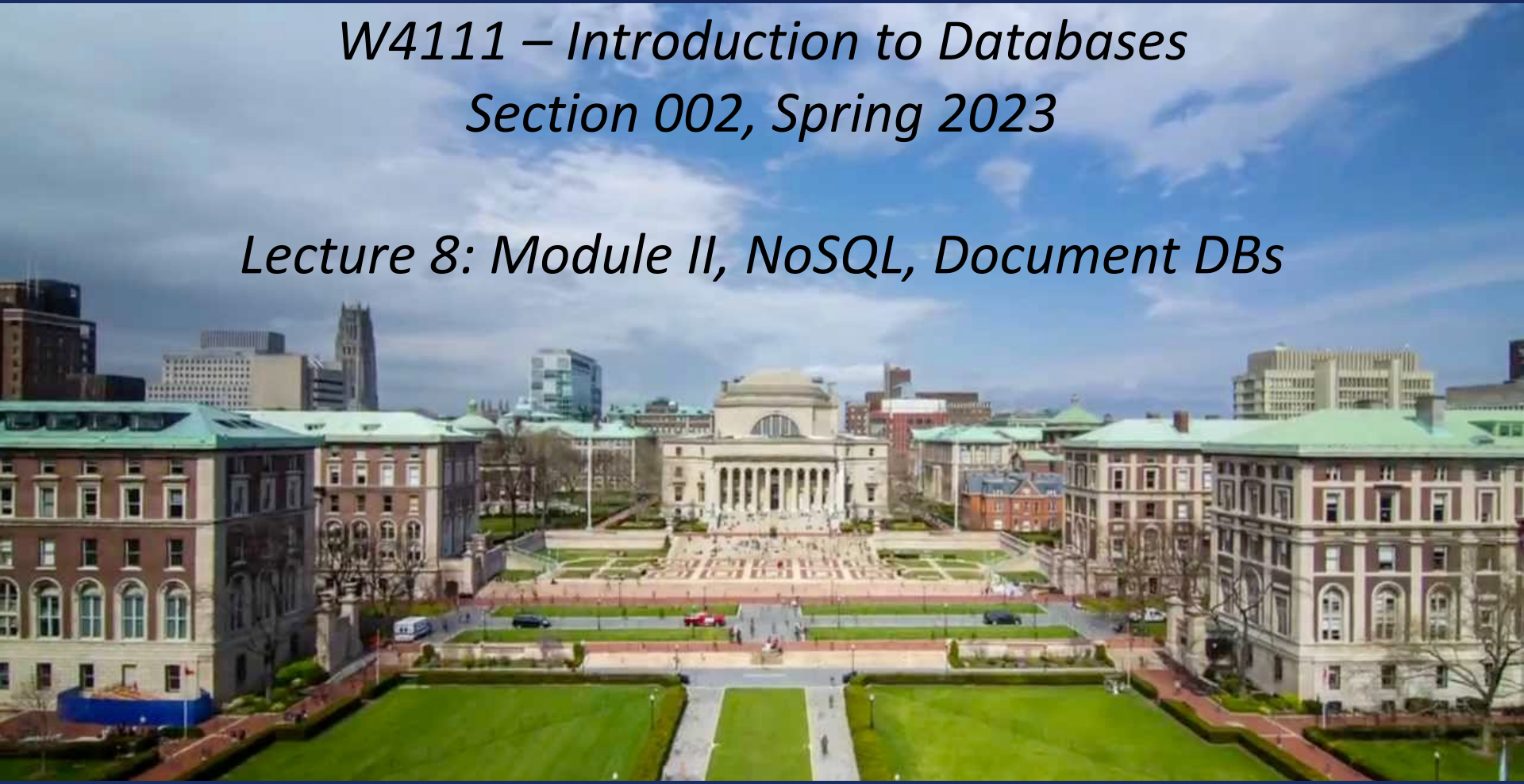
W4111 – Introduction to Databases
Section 002, Spring 2023

Lecture 8: Module II, NoSQL, Document DBs



Contents

Module II Kickoff

Course Modules – Reminder

Course Overview

Each section of W4111 is slightly different based on student interest and professor's focus. There is a common, core syllabus. Professors cover topics in different orders and grouping based on teaching style.

This section of W4111 has four modules:

- **Foundational concepts (50% of semester):** This module covers concepts like data models, relational model, relational databases and applications, schema, normalization, ... The module focuses on the relational model and relational databases. The concepts are critical and foundational for all types of databases and data centric applications.
- **Database management system architecture and implementation (10%):** This module covers the software architecture, algorithms and implementation techniques that allow databases management systems to deliver functions. Topics include memory hierarchy, storage systems, caching/buffer pools, indexes, query processing, query optimization, transaction processing, isolation and concurrency control.
- **NoSQL – "Not Only SQL" databases (20%):** This module provides motivation for "NoSQL" data models and databases, and covers examples and use cases. The module also includes cloud databases and databases-as-a-service.
- **Data Enabled Decision Support (20%):** This module covers data warehouses, data import and cleanse, OLAP, Pivot Tables, Star Schema, reporting and visualization, and provides and overview of analysis techniques, e.g. clustering, classification, analysis, mining.

Module II – DBMS Architecture and Implementation Overview and Reminder

Module II – DBMS Architecture and Implementation

What is a database? In essence a database is nothing more than a collection of information that exists over a long period of time, often many years. In common parlance, the term *database* refers to a collection of data that is managed by a DBMS. The DBMS is expected to:

1. Allow users to create new databases and specify their *schemas* (logical structure of the data), using a specialized *data-definition language*.

Covered for the relational model.

Database Systems: The Complete Book (2nd Edition)
by Hector Garcia-Molina (Author), Jeffrey D. Ullman (Author), Jennifer Widom (Author)

Module II – DBMS Architecture and Implementation

2. Give users the ability to *query* the data (a "query" is database lingo for a question about the data) and modify the data, using an appropriate language, often called a *query language* or *data-manipulation language*.

3. Support the storage of very large amounts of data — many terabytes or more — over a long period of time, allowing efficient access to the data for queries and database modifications.

4. Enable *durability*, the recovery of the database in the face of failures, errors of many kinds, or intentional misuse.

5. Control access to data from many users at once, without allowing unexpected interactions among users (called *isolation*) and without actions on the data to be performed partially but not completely (called *atomicity*).

Database Systems: The Complete Book (2nd Edition)
by Hector Garcia-Molina (Author), Jeffrey D. Ullman (Author), Jennifer Widom (Author)

Purpose of Database Systems

In the early days, database applications were built directly on top of file systems, which leads to:

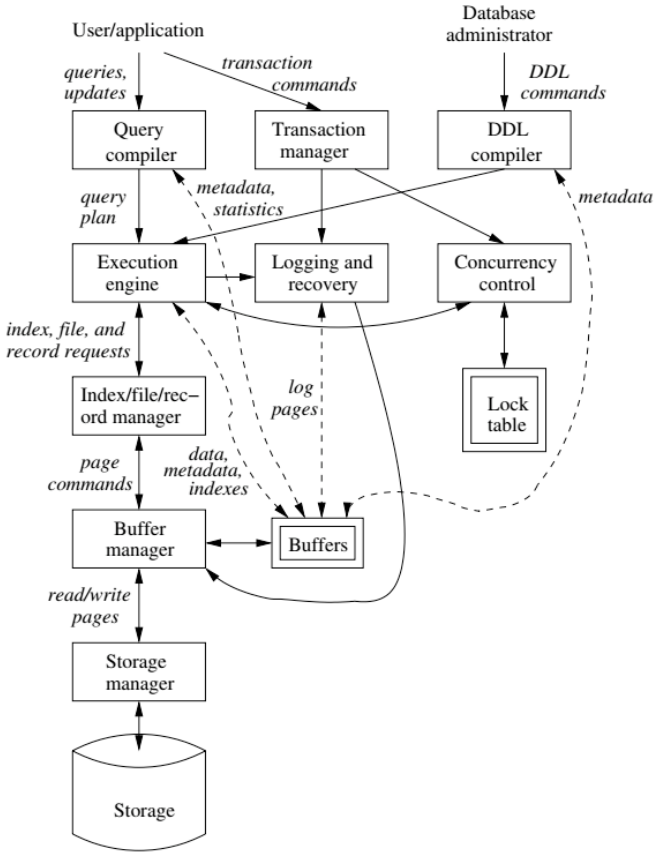
- Data redundancy and inconsistency: data is stored in multiple file formats resulting in duplication of information in different files
- Difficulty in accessing data
  - Need to write a new program to carry out each new task
- Data isolation
  - Multiple files and formats
- Integrity problems
  - Integrity constraints (e.g., account balance > 0) become "buried" in program code rather than being stated explicitly
  - Hard to add new constraints or change existing ones

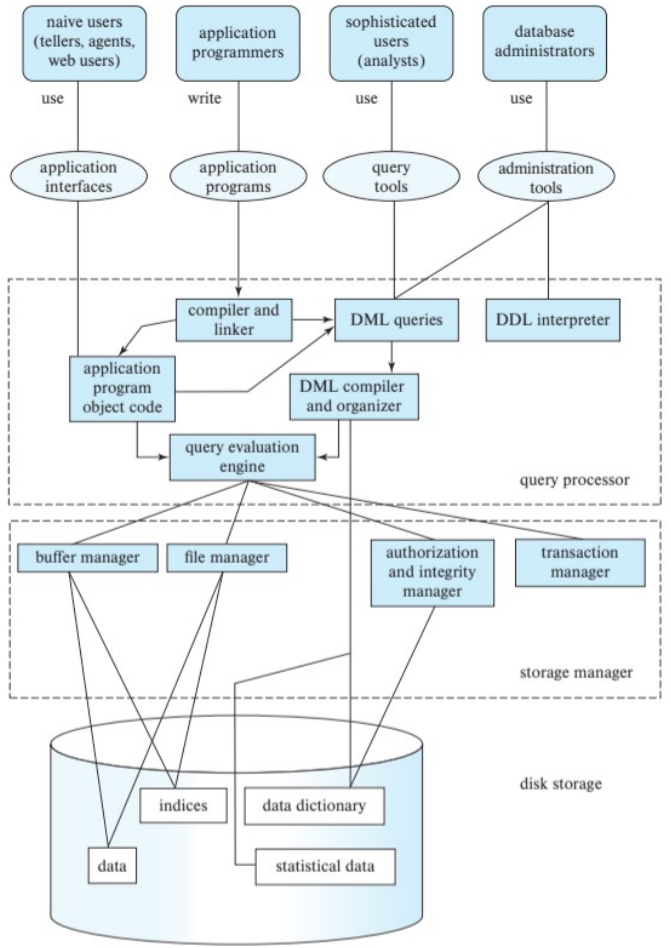
Purpose of Database Systems (Cont.)

- Atomicity of updates
  - Failures may leave database in an inconsistent state with partial updates carried out
  - Example: Transfer of funds from one account to another should either complete or not happen at all
- Concurrent access by multiple users
  - Concurrent access needed for performance
  - Uncontrolled concurrent accesses can lead to inconsistencies
    - Ex: Two people reading a balance (say 100) and updating it by withdrawing money (say 50 each) at the same time
- Security problems
  - Hard to provide user access to some, but not all, data

Database systems offer solutions to all the above problems. In module I, we explored how users interact with the (some) of the functions through DDL and DML. In module II, we will explore *how* DBMS implement the capabilities "under the covers."

DBMS Arch.

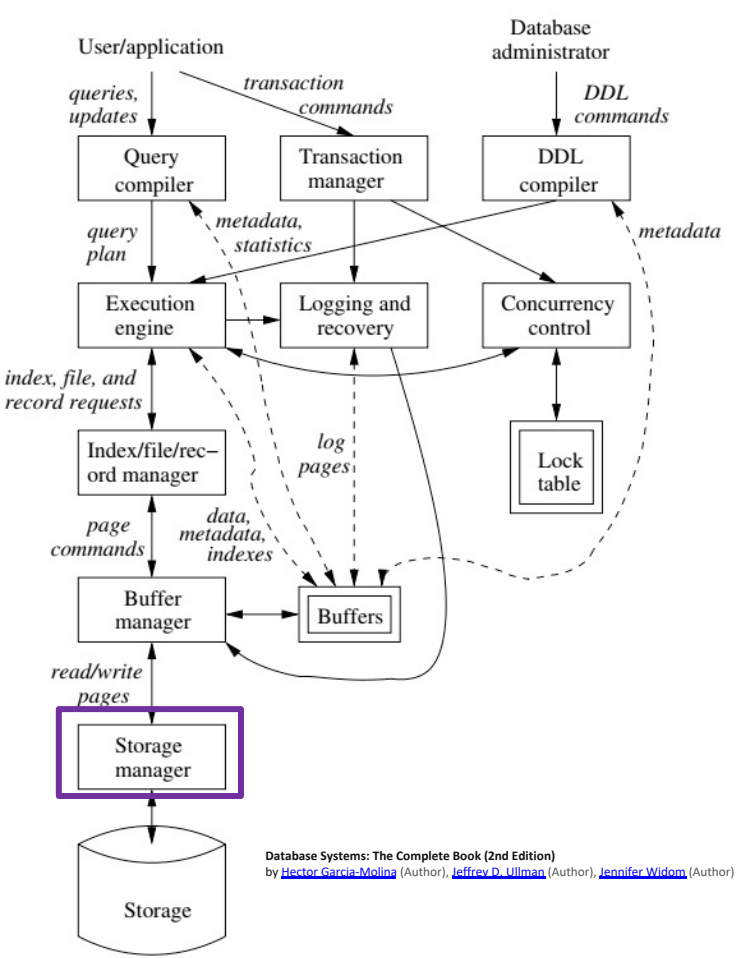




Data Management

Today

- Load/save things quickly.



Database Systems: The Complete Book (2nd Edition)
by Hector Garcia-Molina (Author), Jeffrey D. Ullman (Author), Jennifer Widom (Author)

Disks
Input/Output (IO)

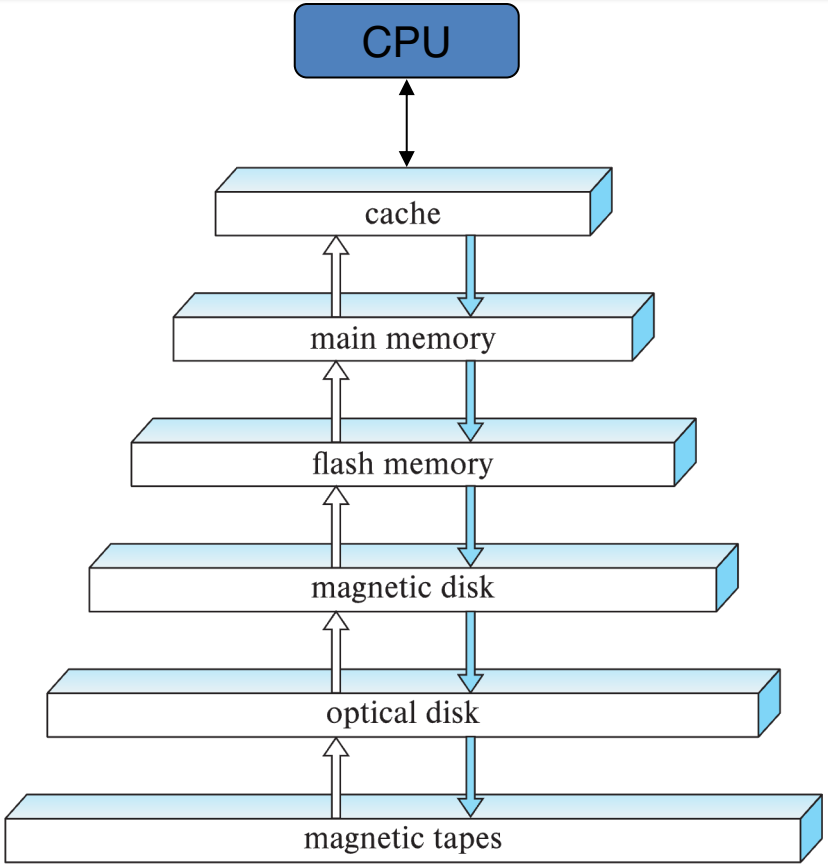
Disks as Far as the Eye can See

Classification of Physical Storage Media

- Can differentiate storage into:
  - **volatile storage:** loses contents when power is switched off
  - **non-volatile storage**:
    - Contents persist even when power is switched off.
    - Includes secondary and tertiary storage, as well as batter-backed up main-memory.
- Factors affecting choice of storage media include
  - Speed with which data can be accessed
  - Cost per unit of data
  - Reliability

From: Database System Concepts, 7th Ed.

Storage Hierarchy



From: Database System Concepts, 7th Ed.

Memory Hierarchy

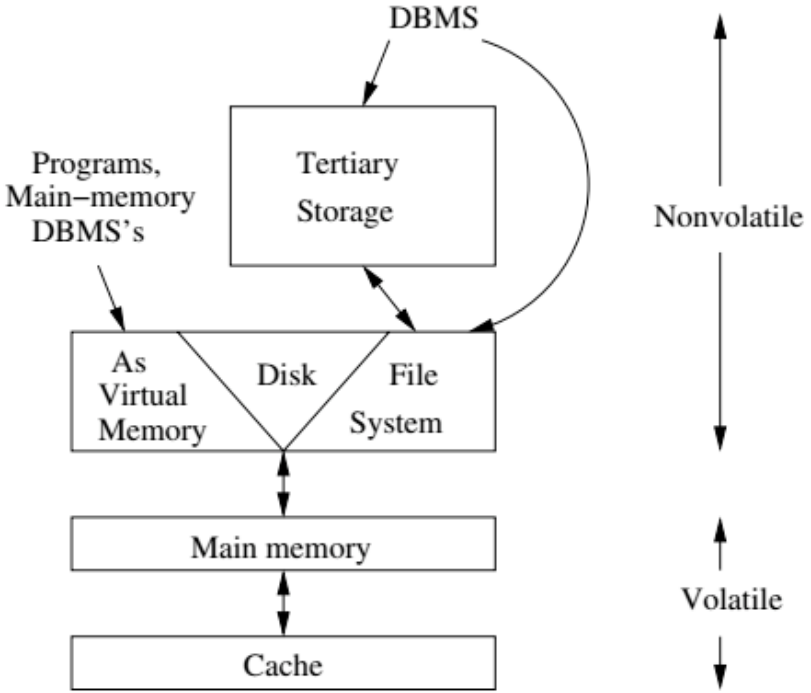


Figure 13.1: The memory hierarchy

Slower
Cheaper
Unlimited

Faster
More expensive
Limited

From: Database System Concepts, 7th Ed.

Memory Hierarchy (Very Old Numbers – Still Directionally Valid)

Storage Technology

Price, Performance & Capacity

Technologies	Capacity (GB)	Latency (microS)	IOPs	Cost/IOPS ($)	Cost/GB ($)
Cloud Storage	*Unlimited*	*60,000*	*20*	*17c/GB*	*0.15/month*
Capacity HDDs	*2,500*	*12,000*	*250*	*1.67*	*0.15*
Performance HDDs	*300*	*7,000*	*500*	*1.52*	*1.30*
SSDs (write)	*64*	*300*	*5000*	*0.20*	*13*
SSDs (read only)	*64*	*45*	*30,000*	*0.03*	*13*
DRAM	*8*	*0.005*	*500,000*	*0.001*	*52*

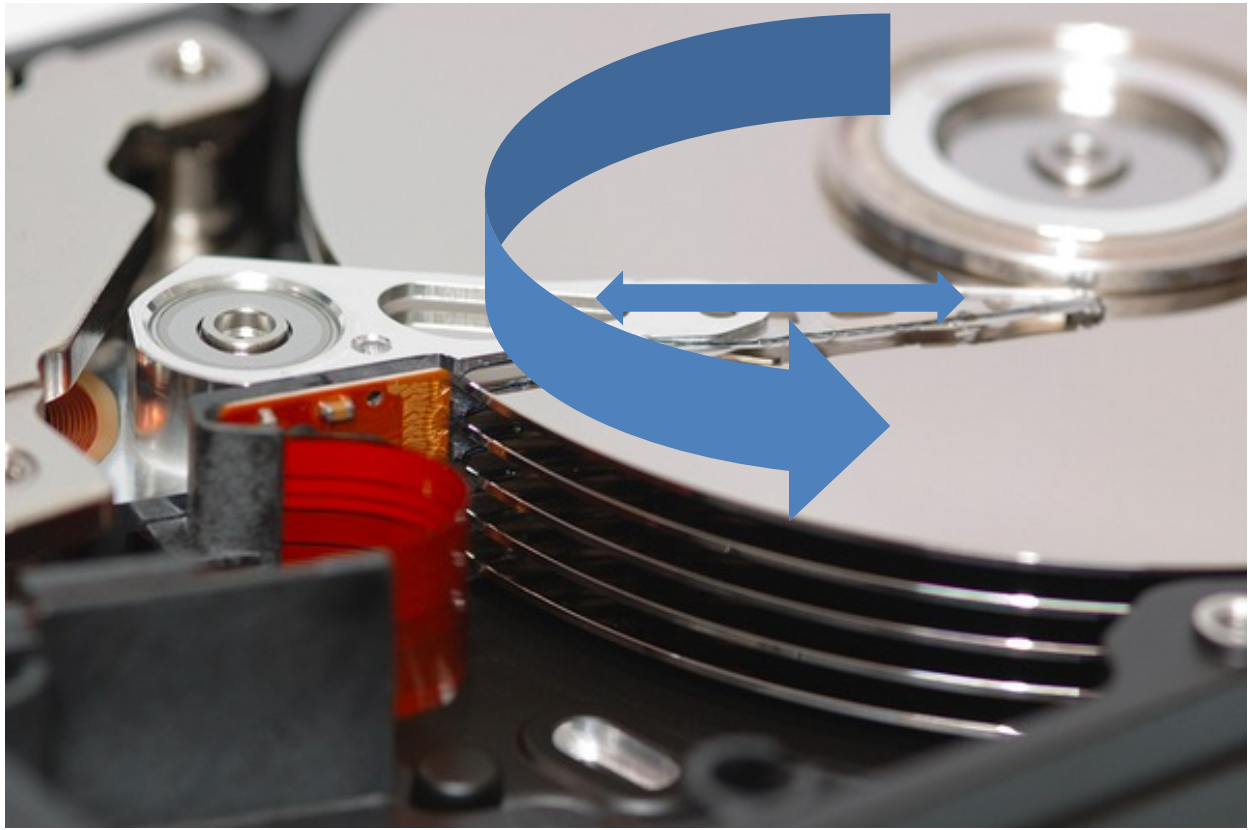
- These numbers are ancient.
- Looking for more modern numbers.
- But, does give an idea of
  - Price
  - Performance
- The general observation is that
  - Performance goes up 10X/level.
  - Price goes up 10x per level.
- Note: One major change is improved price performance of SDD relative to HDD for large data.

Storage Hierarchy (Cont.)

- **primary storage:** Fastest media but volatile (cache, main memory).
- **secondary storage:** next level in hierarchy, non-volatile, moderately fast access time
  - Also called **on-line storage**
  - E.g., flash memory, magnetic disks
- **tertiary storage:** lowest level in hierarchy, non-volatile, slow access time
  - also called **off-line storage** and used for **archival storage**
  - e.g., magnetic tape, optical storage
  - Magnetic tape
    - Sequential access, 1 to 12 TB capacity
    - A few drives with many tapes
    - Juke boxes with petabytes (1000's of TB) of storage

From: Database System Concepts, 7th Ed.

Hard Disk Drive



Disk Configuration

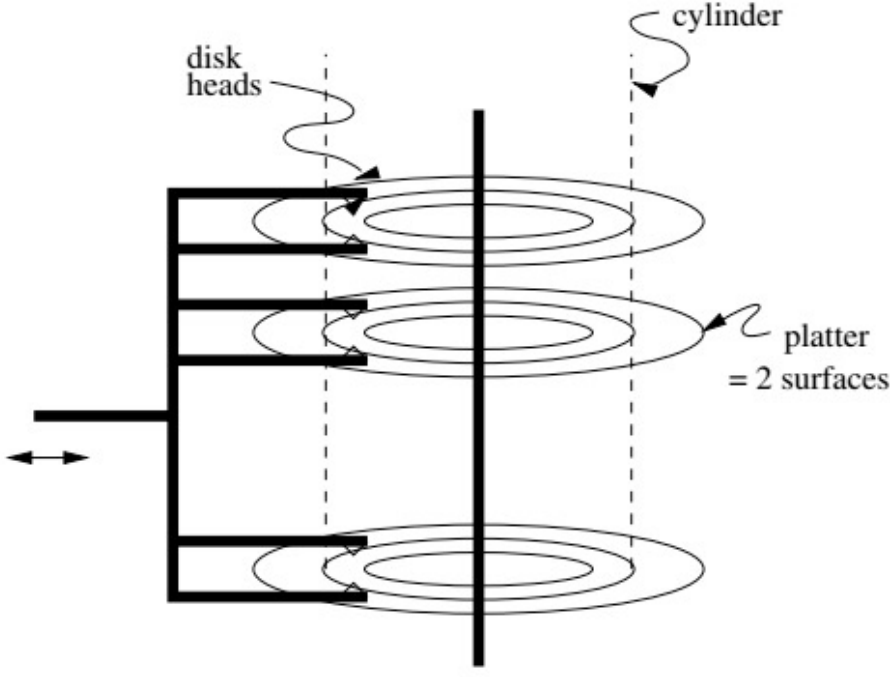


Figure 13.2: A typical disk

Components of disk I/O delay

Seek: Move head to cylinder/track.

Rotation: Wait for sector to get under head

Transfer: Move data from disk to memory.

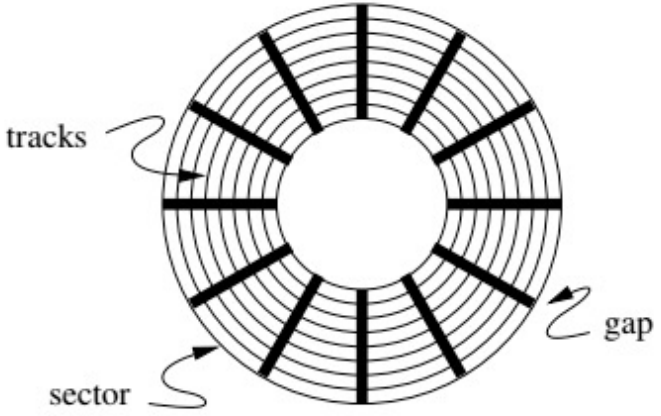
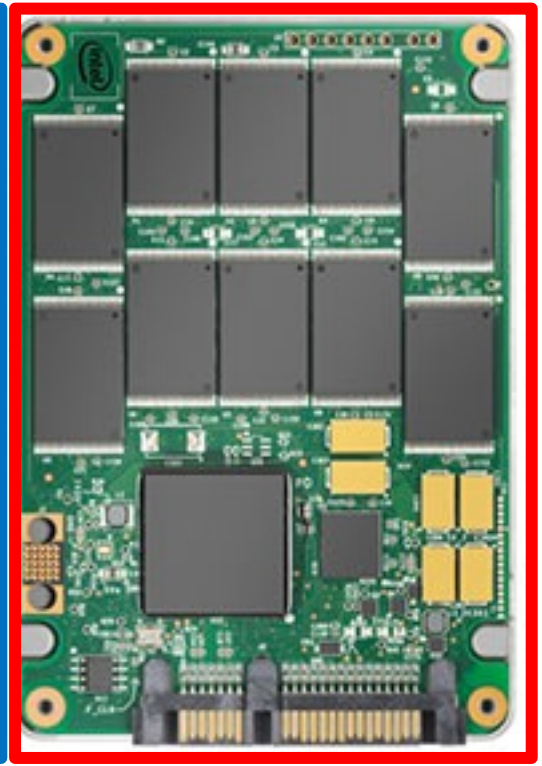


Figure 13.3: Top view of a disk surface

Database Systems: The Complete Book (2nd Edition)
by Hector Garcia-Molina (Author), Jeffrey D. Ullman (Author), Jennifer Widom (Author)

Hard Disk versus Solid State Disk

Hard Disk Drive



Solid State Drive

Flash Storage

From: Database System Concepts, 7th Ed.

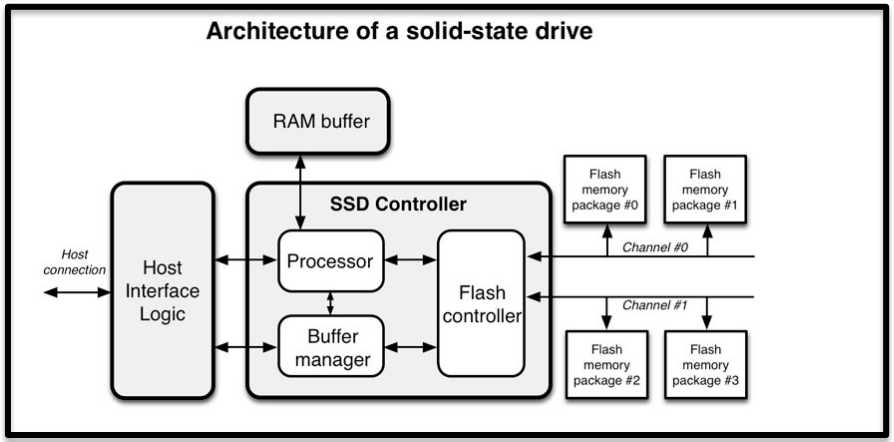
- NOR flash vs NAND flash
- NAND flash
  - used widely for storage, cheaper than NOR flash
  - requires page-at-a-time read (page: 512 bytes to 4 KB)
    - 20 to 100 microseconds for a page read
    - Not much difference between sequential and random read
  - Page can only be written once
    - Must 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

Despite the radically different implementation, it has a disk oriented API.

Logical Block Addressing

- Concept:
  - The *unit of transfer* from a “disk” to the computers memory is a “block. Blocks are usually relatively large, e.g. 16 KB, 32 KB, ...
  - A program that reads or write a single byte, requires the database engine (or file system) to read/write the entire block.
- The address of a block in the entire space of blocks is:
  - (Device ID, Block ID)
  - Block ID is simple 0, 1, 2, ...
- The disk controller and disk implementation translate the *logical block address* into the *physical address of blocks.*
- The physical address changes over time for various reasons, e.g. performance optimization, internal HW failure, etc.

Logical/Physical Block Addressing



Read/Write N

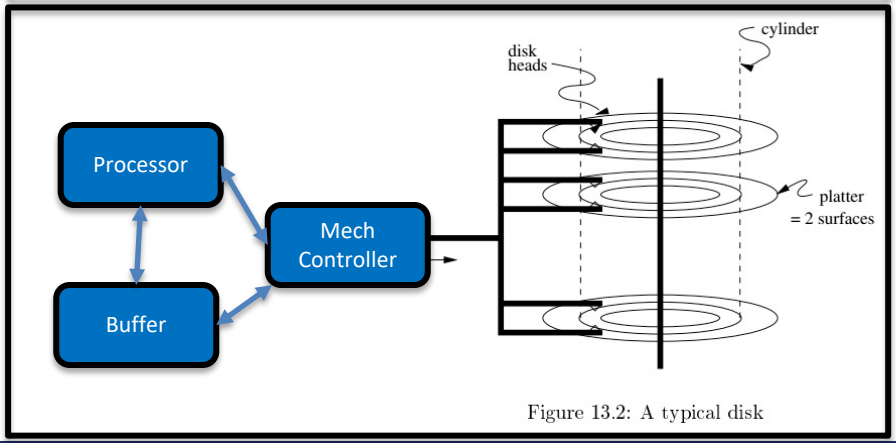


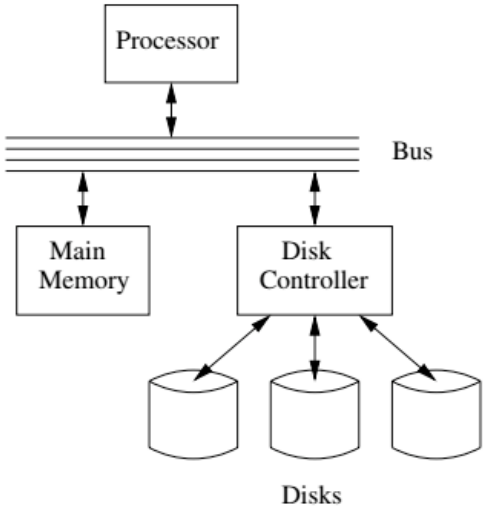
Figure 13.2: A typical disk

Read/Write N

The mapping from LBA to physical block address can change over time.

- Internal HW failure.
- SSD writes in a funny way.
  - You have to erase before writing.
  - So, the SSD (for performance)
    - Writes to an empty block.
    - Erase the original block.
- Performance optimization on HDD
  - Based on block access patterns.
  - Place blocks on cylinder/sector/head in a way to minimize:
    - Seek
    - Rotate

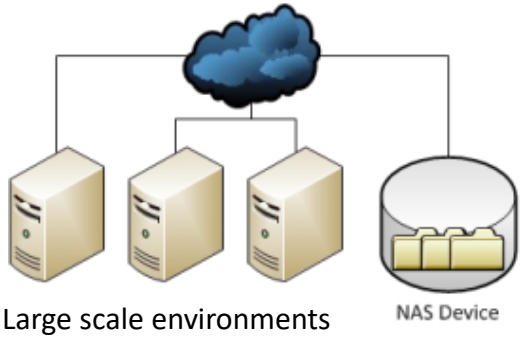
I/O Architecture



How we normally think of disks and I/O.

Network Attached Storage

- Shared storage over shared network
- File system
- Easier management

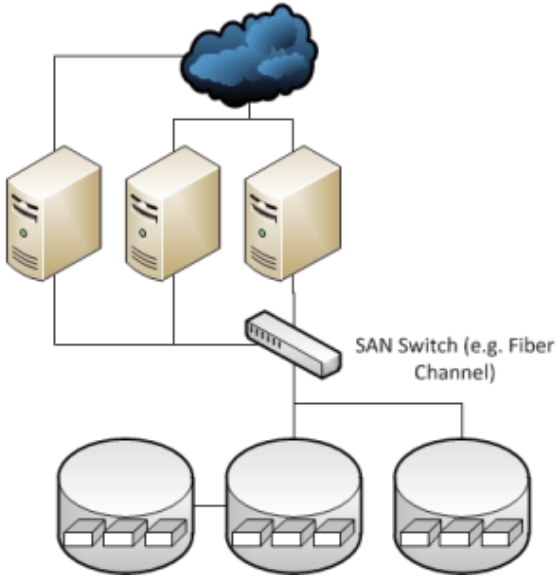


Large scale environments

- The bus-controller connection is over some kind of network.
- The disk controller is at the disks, and basically a "computer" with SW.
- Network is either
  - Standard communication network, or
  - Highly optimized I/O network.

Storage Area Network

- Shared storage over dedicated network
- Raw storage
- Fast, but costly



Magnetic Disks

- **Read-write head**
- Surface of platter divided into circular **tracks**
  - Over 50K-100K tracks per platter on typical hard disks
- Each track is divided into **sectors.**
  - A sector is the smallest unit of data that can be read or written.
  - Sector size typically 512 bytes
  - Typical sectors per track: 500 to 1000 (on inner tracks) to 1000 to 2000 (on outer tracks)
- To read/write a sector
  - disk arm swings to position head on right track
  - platter spins continually; data is read/written as sector passes under head
- Head-disk assemblies
  - multiple disk platters on a single spindle (1 to 5 usually)
  - one head per platter, mounted on a common arm.
- **Cylinder** *i* consists of i^{th} track of all the platters

From: Database System Concepts, 7th Ed.

Magnetic Disks (Cont.)

- **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
  - Computes and attaches **checksums** to each sector to verify that data is read back correctly
    - If data is corrupted, with very high probability stored checksum won't match recomputed checksum
  - Ensures successful writing by reading back sector after writing it
  - Performs **remapping of bad sectors**

From: Database System Concepts, 7th Ed.

Performance Measures of Disks

- **Access time** – the time it takes from when a read or write request is issued to when data transfer begins. Consists of:
  - **Seek time** – time it takes to reposition the arm over the correct track.
    - Average seek time is 1/2 the worst case seek time.
      - Would be 1/3 if all tracks had the same number of sectors, and we ignore the time to start and stop arm movement
    - 4 to 10 milliseconds on typical disks
  - **Rotational latency** – time it takes for the sector to be accessed to appear under the head.
    - 4 to 11 milliseconds on typical disks (5400 to 15000 r.p.m.)
    - Average latency is 1/2 of the above latency.
  - Overall latency is 5 to 20 msec depending on disk model
- **Data-transfer rate** – the rate at which data can be retrieved from or stored to the disk.
  - 25 to 200 MB per second max rate, lower for inner tracks

From: Database System Concepts, 7th Ed.

Performance Measures (Cont.)

- **Disk block** is a logical unit for storage allocation and retrieval
  - 4 to 16 kilobytes typically
    - Smaller blocks: more transfers from disk
    - Larger blocks: more space wasted due to partially filled blocks
- **Sequential access pattern**
  - Successive requests are for successive disk blocks
  - Disk seek required only for first block
- **Random access pattern**
  - Successive requests are for blocks that can be anywhere on disk
  - Each access requires a seek
  - Transfer rates are low since a lot of time is wasted in seeks
- **I/O operations per second (IOPS)**
  - Number of random block reads that a disk can support per second
  - 50 to 200 IOPS on current generation magnetic disks

From: Database System Concepts, 7th Ed.

Performance Measures (Cont.)

- **Mean time to failure (MTTF)** – the average time the disk is expected to run continuously without any failure.
  - Typically 3 to 5 years
  - Probability of failure of new disks is quite low, corresponding to a "theoretical MTTF" of 500,000 to 1,200,000 hours for a new disk
    - E.g., an MTTF of 1,200,000 hours for a new disk means that given 1000 relatively new disks, on an average one will fail every 1200 hours
  - MTTF decreases as disk ages

From: Database System Concepts, 7th Ed.

Logical Block Addressing (https://gerardnico.com/wiki/data_storage/lba)

3 - The LBA scheme

LBA	C	H	S
0	0	0	0
1	0	0	1
2	0	0	2
3	0	0	3
4	0	0	4
5	0	0	5
6	0	0	6
7	0	0	7
8	0	0	8
9	0	0	9
10	0	1	0
11	0	1	1
12	0	1	2
13	0	1	3
14	0	1	4
15	0	1	5
16	0	1	6
17	0	1	7
18	0	1	8
19	0	1	9
Cylinder 0			

LBA	C	H	S
20	1	0	0
21	1	0	1
22	1	0	2
23	1	0	3
24	1	0	4
25	1	0	5
26	1	0	6
27	1	0	7
28	1	0	8
29	1	0	9
30	1	1	0
31	1	1	1
32	1	1	2
33	1	1	3
34	1	1	4
35	1	1	5
36	1	1	6
37	1	1	7
38	1	1	8
39	1	1	9
Cylinder 1			

- CHS addresses can be converted to LBA addresses using the following formula:

$$LBA = ((C \times HPC) + H) \times SPT + S - 1$$

where,

- C, H and S are the cylinder number, the head number, and the sector number
- LBA is the logical block address
- HPC is the number of heads per cylinder
- SPT is the number of sectors per track

Devices have configuration and metadata APIs that allow storage manager to

- Map between LBA and CHS
- To optimize block placement
- Based on access patterns, statistics, data schema, etc.

I/O Architecture

Redundant
Array
of
Independent
Disks

(RAID)

RAID (https://en.wikipedia.org/wiki/RAID)

"RAID (redundant array of independent disks) is a data storage virtualization technology that combines multiple physical disk drive components into a single logical unit for the purposes of data redundancy, performance improvement, or both. (…)

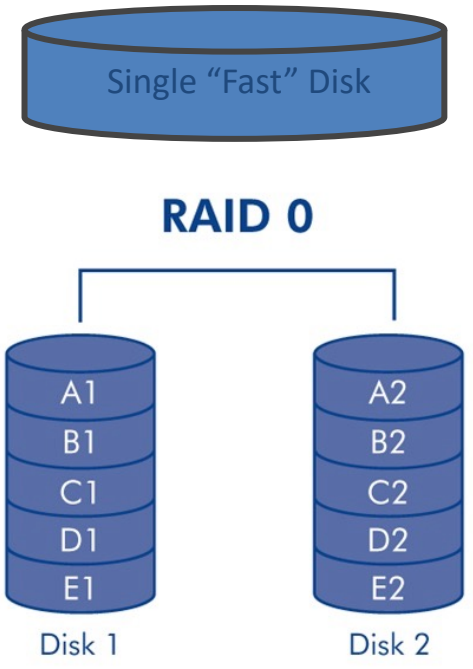
RAID 0 consists of striping, without mirroring or parity. (…)
RAID 1 consists of data mirroring, without parity or striping. (…)
RAID 2 consists of bit-level striping with dedicated Hamming-code parity. (…)
RAID 3 consists of byte-level striping with dedicated parity. (…)
RAID 4 consists of block-level striping with dedicated parity. (…)
RAID 5 consists of block-level striping with distributed parity. (…)
RAID 6 consists of block-level striping with double distributed parity. (…)

Nested RAID

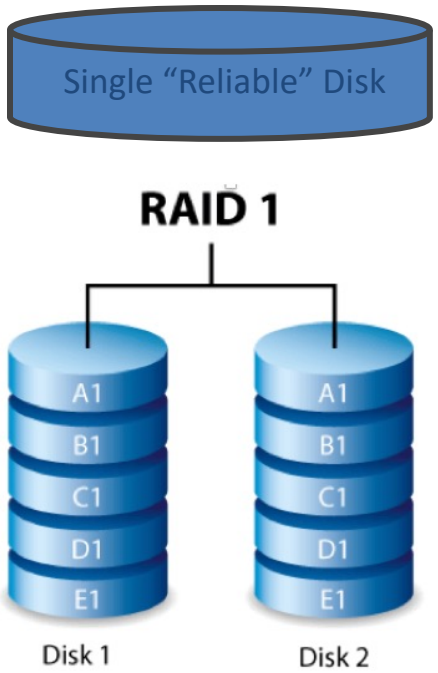
- RAID 0+1: creates two stripes and mirrors them. (…)
- RAID 1+0: creates a striped set from a series of mirrored drives. (…)
- **JBOD RAID N+N:** With JBOD (*just a bunch of disks*), (…)"

RAID-0 and RAID-1

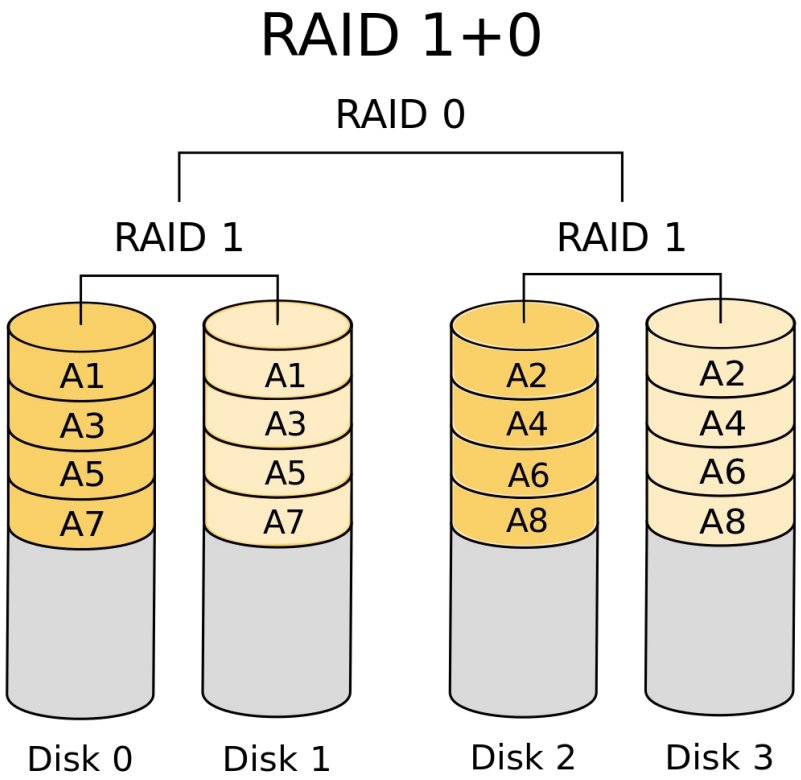
Two physical disks make one single, logical **fast** disk



Two physical disks make one single, logical **reliable** disk



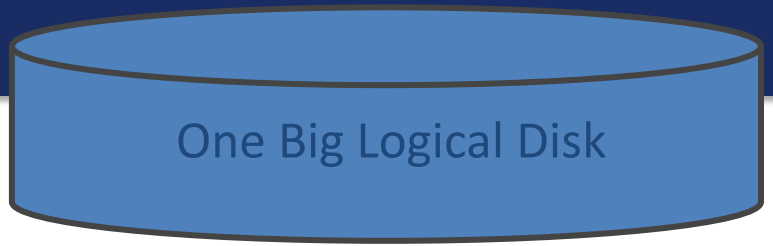
Mixed RAID Modes



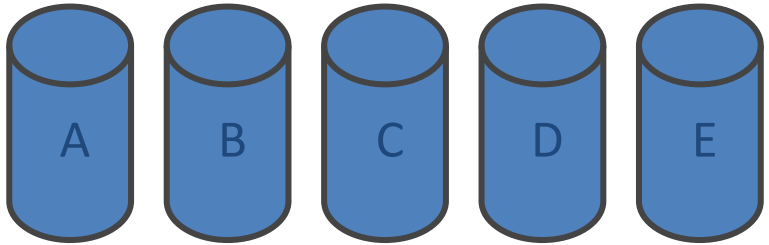
Stripe
And
Mirror

RAID-5

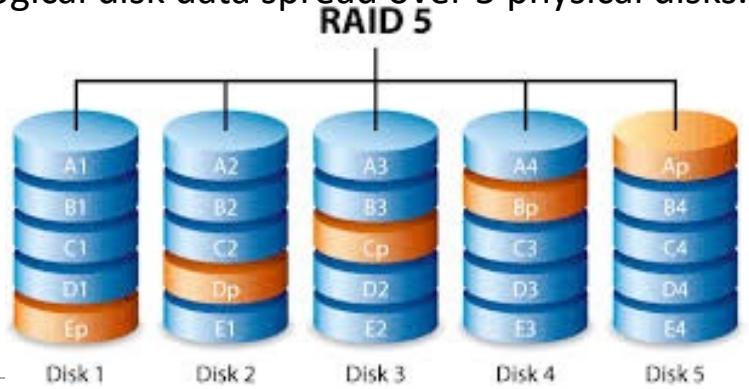
- Improved performance through parallelism
  - Rotation/seek
  - Transfer
- Availability uses *parity blocks*
  - Suppose I have 4 different data blocks on the logical drive A: A1, A2, A3, A4.
  - Parity function: **Ap** = P(A1,A2,A3,A4)
  - Recovery function: A2=R(**Ap**,A1,A3,A4)
- During normal operations:
  - Read processing simply retrieves block.
  - Write processing of A2 updates A2 and Ap
- If an individual disk fails, the RAID
  - Read
    - Continues to function for reads on non-missing blocks.
    - Implements read on missing block by recalculating value.
  - Write
    - Updates block and parity block for non-missing blocks.
    - Computes missing block, and calculates parity based on old and new value.
  - Over time
    - "Hot Swap" the failed disk.
    - Rebuild the missing data from values and parity.



Is actually 5 smaller "logical" disks.



Logical disk data spread over 5 physical disks.



Very Simple Parity Example

- Even-Odd Parity
  - b[i] is an array of bits (0 or 1)
  - P(b[i]) =
    - 0 is an even number of bits = 1. {P([0,1,1,0,1,1])= 0
    - 1 if an odd number of bits = 1. {P(0,0,1,0,1,1]= 1
  - Given an array with one missing bit and the parity bit, I can re-compute the missing bit.
    - Case 1: [0,?,1,0,1,1] has P=0. There must be an EVEN number of ones and ?=1.
    - Case 2: [0,?,1,0,1,1] has P=1. There must be an ODD number of ones and ?=0.
- Block Parity applies this to a set of blocks bitwise
  - A1=[0 1,0,0,1,1]
  - A2=[1 1,1,0,0,0]
  - A3=[0 0,0,1,0,1]
  - Pa=[1 0,1,1,1,0]

➔ If I am missing a block and have the parity block, I can re-compute the missing block bitwise from remaining blocks and parity block.

Data Storage Structures
(Database Systems Concepts, V7, Ch. 13)

File Organization

- The database is stored as a collection of *files*. Each file is a sequence of *records.* A record is a sequence of fields.
- One approach
  - Assume record size is fixed
  - Each file has records of one particular type only
  - Different files are used for different relations

  This case is easiest to implement; will consider variable length records later
- We assume that records are smaller than a disk block

  .

From: Database System Concepts, 7th Ed.

Terminology

- A tuple in a relation maps to a *record*. Records may be
  - *Fixed length*
  - *Variable length*
  - *Variable format* (which we will see in Neo4J, DynamoDB, etc).
- A *block*
  - Is the unit of transfer between disks and memory (buffer pools).
  - Contains multiple records, usually but not always from the same relation.
- The *database address space* contains
  - All of the blocks and records that the database manages
  - Including blocks/records containing data
  - And blocks/records containing free space.

Fixed-Length Records

- Simple approach:
  - Store record *i* starting from byte $n * (i - 1)$, where *n* is the size of each record.
  - Record access is simple but records may cross blocks
    - Modification: do not allow records to cross block boundaries

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

From: Database System Concepts, 7th Ed.

Fixed-Length Records

- Deletion of record *i:* alternatives*:*
  - **move records *i* + 1, . . ., *n* to *i*, . . . , *n* – 1**
  - move record *n* to *i*
  - do not move records, but link all free records on a *free list*

Record 3 deleted

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

From: Database System Concepts, 7th Ed.

Fixed-Length Records

- Deletion of record *i:* alternatives*:*
  - move records *i* + 1, . . ., *n* to *i*, . . . , *n* – 1
  - **move record *n* to *i***
  - do not move records, but link all free records on a *free list*

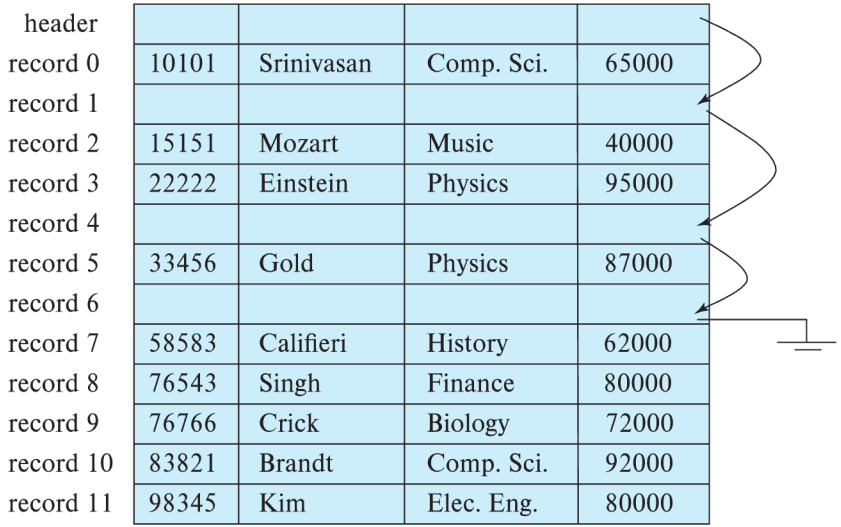
Record 3 deleted and replaced by record 11

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 11	98345	Kim	Elec. Eng.	80000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000

From: Database System Concepts, 7th Ed.

Fixed-Length Records

- Deletion of record *i:* alternatives*:*
  - move records *i* + 1, . . ., *n* to *i*, . . . , *n* – 1
  - move record *n* to *i*
  - **do not move records, but link all free records on a *free list***

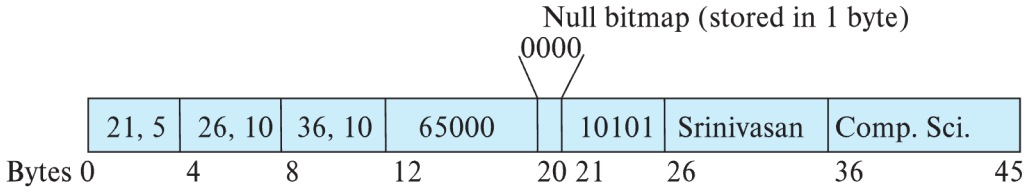


header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

From: Database System Concepts, 7th Ed.

Variable-Length Records

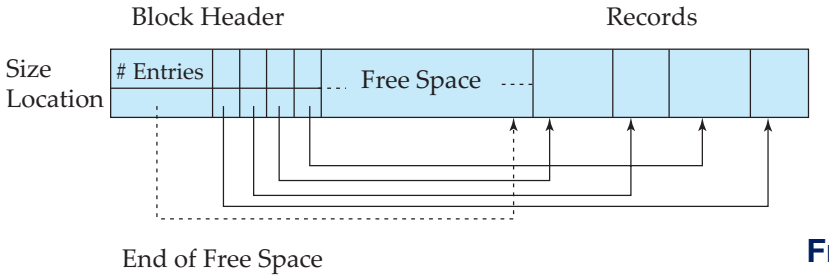
- Variable-length records arise in database systems in several ways:
  - Storage of multiple record types in a file.
  - Record types that allow variable lengths for one or more fields such as strings (**varchar**)
  - Record types that allow repeating fields (used in some older data models).
- Attributes are stored in order
- Variable length attributes represented by fixed size (offset, length), with actual data stored after all fixed length attributes
- Null values represented by null-value bitmap



From: Database System Concepts, 7th Ed.

Variable-Length Records: Slotted Page Structure

- **Slotted page** header contains:
  - number of record entries
  - end of free space in the block
  - location and size of each record
- Records can be moved around within a page to keep them contiguous with no empty space between them; entry in the header must be updated.
- Pointers should not point directly to record — instead they should point to the entry for the record in header.



From: Database System Concepts, 7th Ed.

Storing Large Objects

- E.g., blob/clob types
- Records must be smaller than pages
- Alternatives:
  - Store as files in file systems
  - Store as files managed by database
  - Break into pieces and store in multiple tuples in separate relation
    - PostgreSQL TOAST

From: Database System Concepts, 7th Ed.

Organization of Records in Files

- **Heap** – record can be placed anywhere in the file where there is space
- **Sequential** – store records in sequential order, based on the value of the search key of each record
- In a **multitable clustering file organization** records of several different relations can be stored in the same file
  - Motivation: store related records on the same block to minimize I/O
- **B^+-tree file organization**
  - Ordered storage even with inserts/deletes
  - More on this in Chapter 14
- **Hashing** – a hash function computed on search key; the result specifies in which block of the file the record should be placed
  - More on this in Chapter 14

From: Database System Concepts, 7th Ed.

Heap File Organization

- Records can be placed anywhere in the file where there is free space
- Records usually do not move once allocated
- Important to be able to efficiently find free space within file
- **Free-space map**
  - Array with 1 entry per block. Each entry is a few bits to a byte, and records fraction of block that is free
  - In example below, 3 bits per block, value divided by 8 indicates fraction of block that is free

4	2	1	4	7	3	6	5	1	2	0	1	1	0	5	6

  - Can have second-level free-space map
  - In example below, each entry stores maximum from 4 entries of first-level free-space map

4	7	2	6

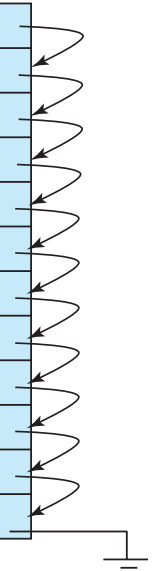
- Free space map written to disk periodically, OK to have wrong (old) values for some entries (will be detected and fixed)

From: Database System Concepts, 7th Ed.

Sequential File Organization

- Suitable for applications that require sequential processing of the entire file
- The records in the file are ordered by a search-key

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

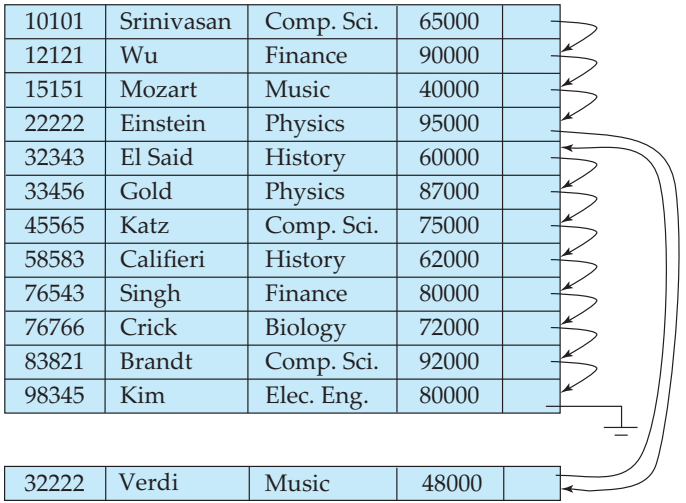


From: Database System Concepts, 7th Ed.

Sequential File Organization (Cont.)

From: Database System Concepts, 7th Ed.

- Deletion – use pointer chains
- Insertion –locate the position where the record is to be inserted
  - if there is free space insert there
  - if no free space, insert the record in an overflow block
  - In either case, pointer chain must be updated
- Need to reorganize the file from time to time to restore sequential order



10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

32222	Verdi	Music	48000	

Partitioning

- **Table partitioning**: Records in a relation can be partitioned into smaller relations that are stored separately
- E.g., *transaction* relation may be partitioned into *transaction_2018, transaction_2019, etc.*
- Queries written on *transaction* must access records in all partitions
  - Unless query has a selection such as *year*=2019, in which case only one partition in needed
- Partitioning
  - Reduces costs of some operations such as free space management
  - Allows different partitions to be stored on different storage devices
    - E.g., *transaction* partition for current year on SSD, for older years on magnetic disk

Column-Oriented Storage

- Also known as **columnar representation**
- Store each attribute of a relation separately
- Example

10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

From: Database System Concepts, 7th Ed.

Columnar Representation

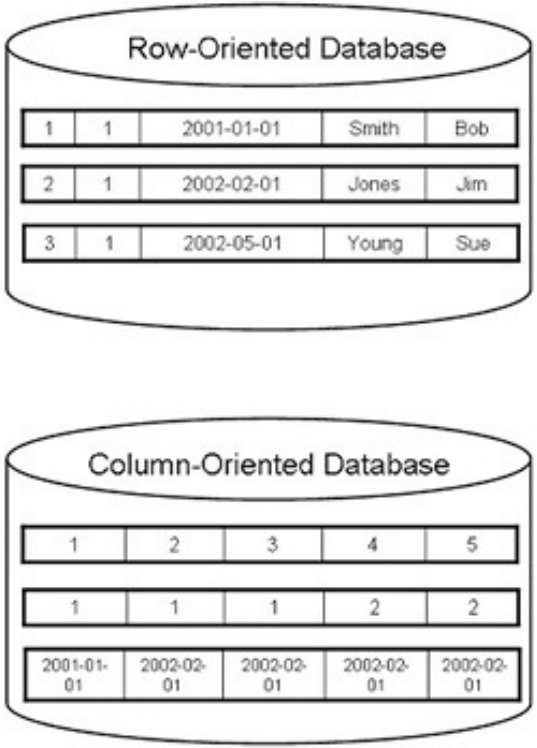
- Benefits:
  - Reduced IO if only some attributes are accessed
  - Improved CPU cache performance
  - Improved compression
  - **Vector processing** on modern CPU architectures
- Drawbacks
  - Cost of tuple reconstruction from columnar representation
  - Cost of tuple deletion and update
  - Cost of decompression
- Columnar representation found to be more efficient for decision support than row-oriented representation
- Traditional row-oriented representation preferable for transaction processing
- Some databases support both representations
  - Called **hybrid row/column stores**

From: Database System Concepts, 7th Ed.

Row vs Column

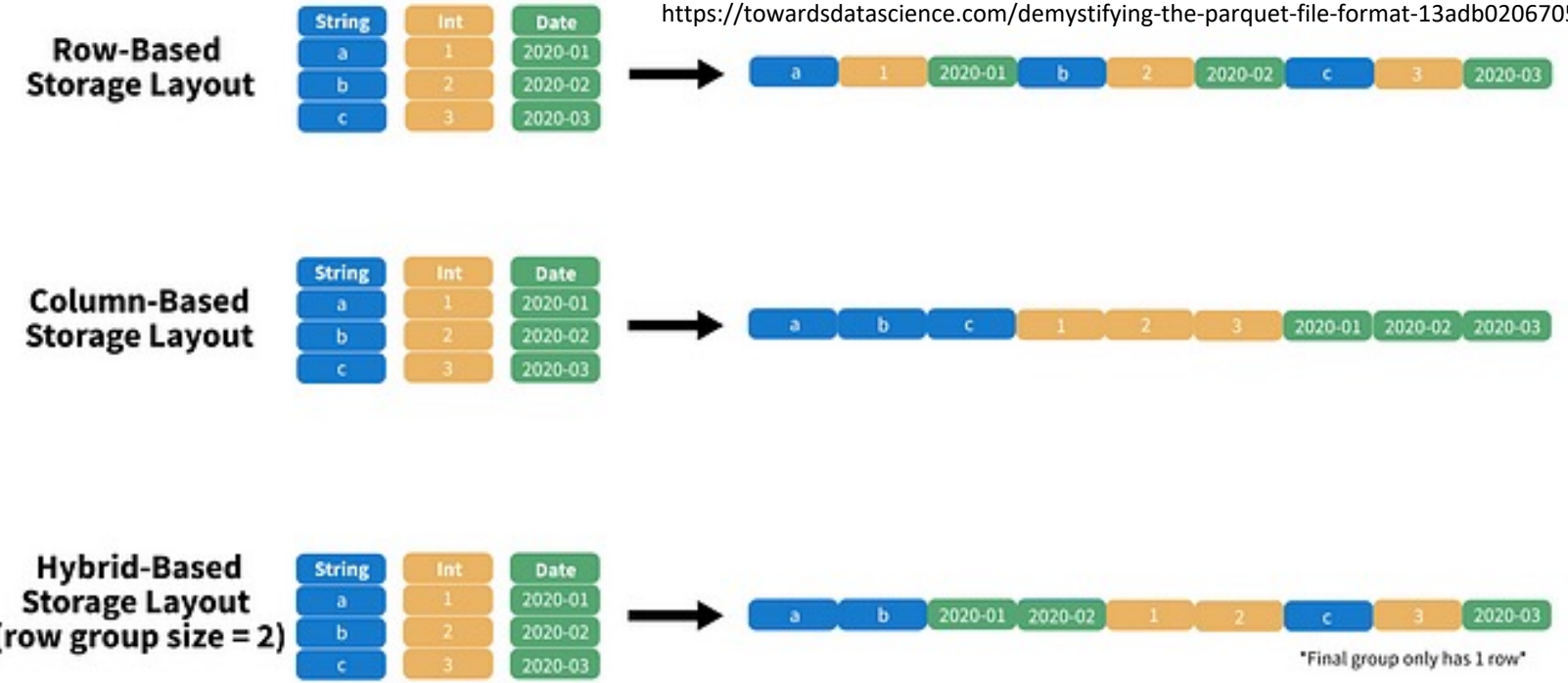
- Columnar and Row are both
  - Relational
  - Support SQL operations
- But differ in data storage
  - Row keeps row data together in blocks.
  - Columnar keys column data together in blocks.
- This determines performance for different types of query, e.g.
  - Columnar is extremely powerful for BI scenarios
    - Aggregation ops, e.g. SUM, AVG
    - PROJECT (do not load all of the row) to get a few columns
  - Row is powerful for OLTP. Transaction typically create and retrieve
    - One row at a time
    - All the columns of a single row.

Emp_no	Dept_id	Hire_date	Emp_ln	Emp_fn
1	1	2001-01-01	Smith	Bob
2	1	2002-02-01	Jones	Jim
3	1	2002-05-01	Young	Sue
4	2	2003-02-01	Stemle	Bill
5	2	1999-06-15	Aurora	Jack
6	3	2000-08-15	Jung	Laura



Columnar File Representation

https://towardsdatascience.com/demystifying-the-parquet-file-format-13adb0206705



Apache parquet is an open-source file format that provides efficient storage and fast read speed. It uses a hybrid storage format which sequentially stores chunks of columns, lending to high performance when selecting and filtering data. On top of strong compression algorithm support (snappy, gzip, LZO), it also provides some clever tricks for reducing file scans and encoding repeat variables.

NoSQL

Concepts

Overview (I) (https://en.wikipedia.org/wiki/NoSQL)

A **NoSQL** (originally referring to "non SQL" or "non relational")[1] database provides a mechanism for storage and retrieval of data that is modeled in means other than the tabular relations used in relational databases. Such databases have existed since the late 1960s, but did not obtain the "NoSQL" moniker until a surge of popularity in the early twenty-first century,[2] triggered by the needs of Web 2.0 companies such as Facebook, Google, and Amazon.com.[3][4][5] NoSQL databases are increasingly used in big data and real-time web applications.[6] NoSQL systems are also sometimes called "Not only SQL" to emphasize that they may support SQL-like query languages.[7][8]

Motivations for this approach include: simplicity of design, simpler "horizontal" scaling to clusters of machines (which is a problem for relational databases),[2] and finer control over availability. The data structures used by NoSQL databases (e.g. key-value, wide column, graph, or document) are different from those used by default in relational databases, making some operations faster in NoSQL. The particular suitability of a given NoSQL database depends on the problem it must solve. Sometimes the data structures used by NoSQL databases are also viewed as "more flexible" than relational database tables.[9]

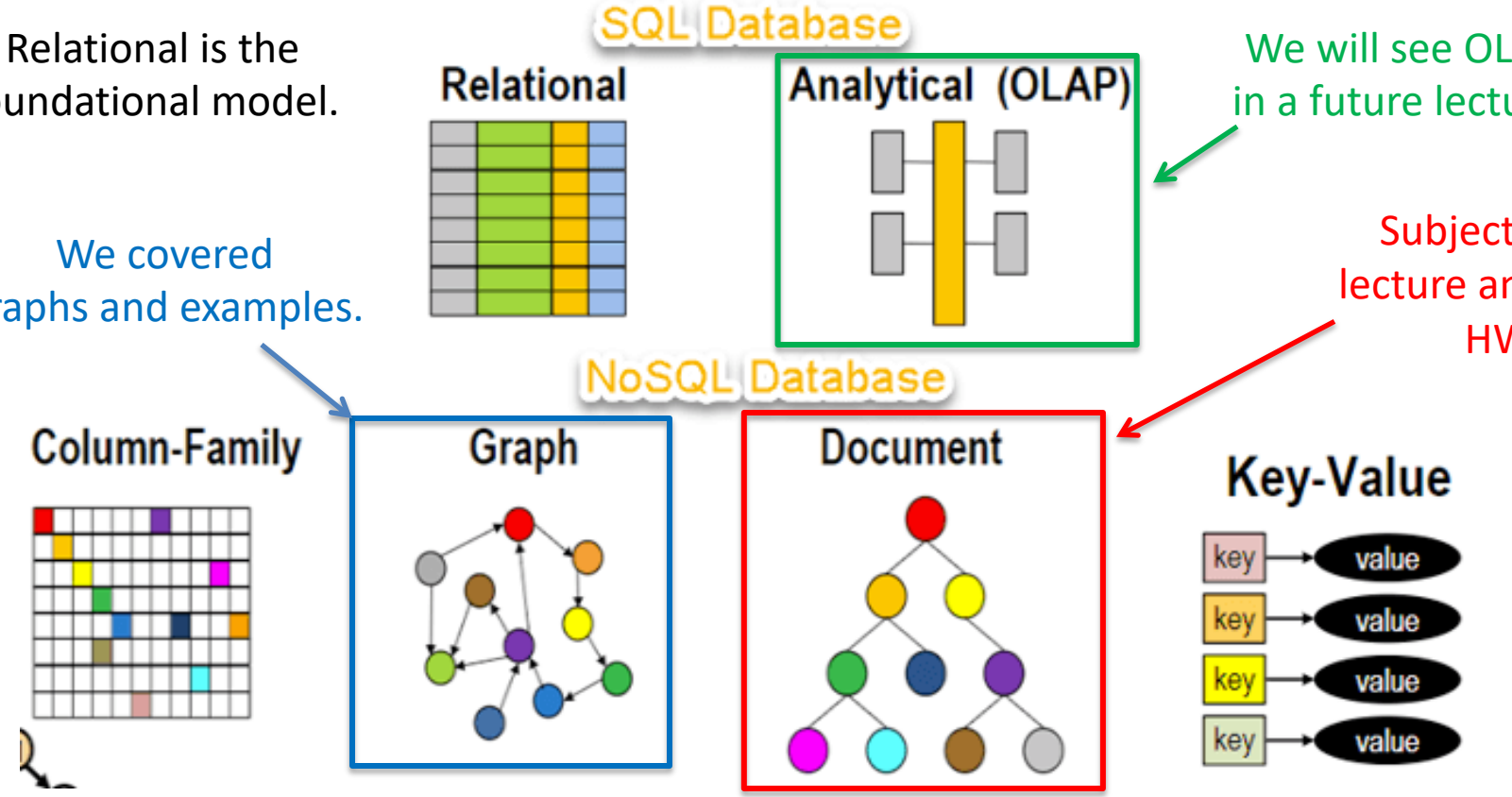
Overview (I) (https://en.wikipedia.org/wiki/NoSQL)

Many NoSQL stores compromise consistency (in the sense of the CAP theorem) in favor of availability, partition tolerance, and speed. Barriers to the greater adoption of NoSQL stores include the use of low-level query languages (instead of SQL, for instance the lack of ability to perform ad-hoc joins across tables), lack of standardized interfaces, and huge previous investments in existing relational databases.[10] Most NoSQL stores lack true ACID transactions, … …

Instead, most NoSQL databases offer a concept of "eventual consistency" in which database changes are propagated to all nodes "eventually" (typically within milliseconds) so queries for data might not return updated data immediately or might result in reading data that is not accurate, a problem known as stale reads.[11] Additionally, some NoSQL systems may exhibit lost writes and other forms of data loss.[12] Fortunately, some NoSQL systems provide concepts such as write-ahead logging to avoid data loss.[13] For distributed transaction processing across multiple databases, data consistency is an even bigger challenge that is difficult for both NoSQL and relational databases. Even current relational databases "do not allow referential integrity constraints to span databases."[14]

Simplistic Classification (https://medium.com/swlh/4-types-of-nosql-databases-d88ad21f7d3b)

Relational is the foundational model.



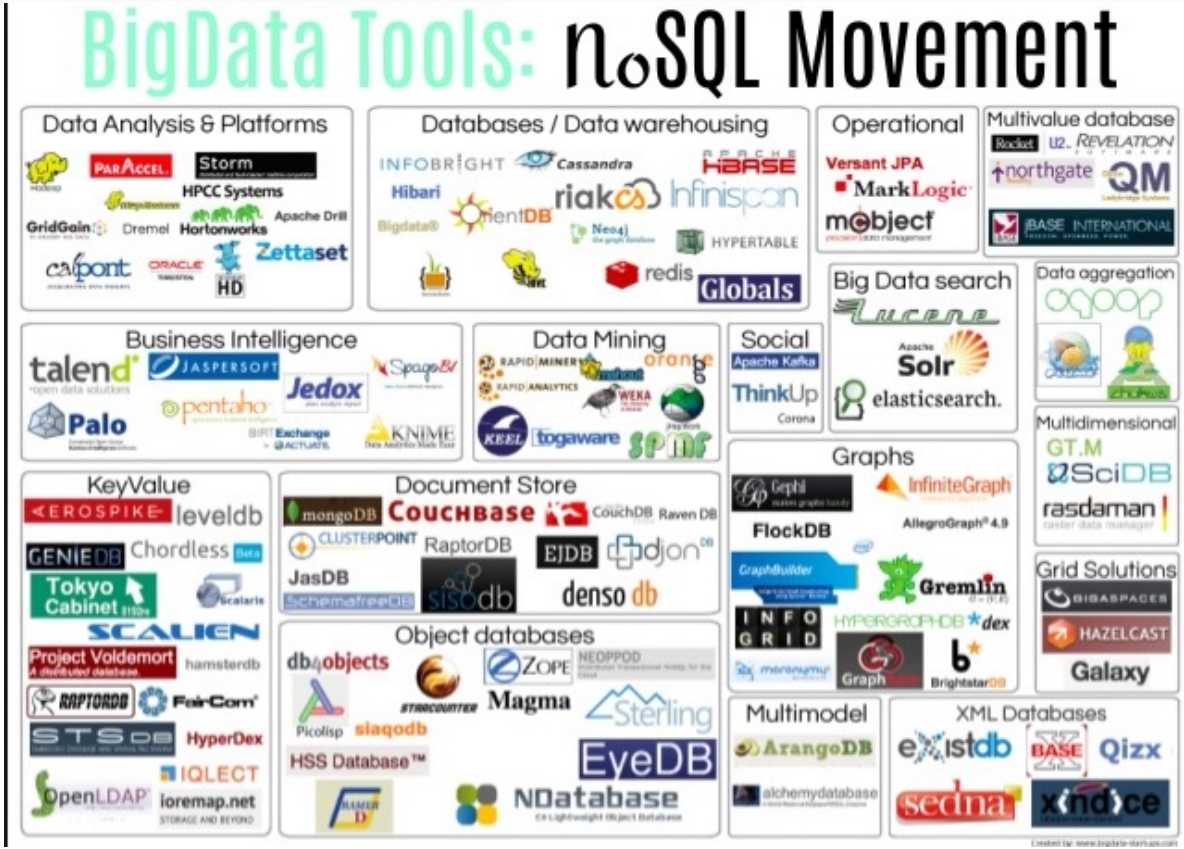
We will see OLAP in a future lecture.

We covered graphs and examples.

Subject of this lecture and part of HW4

One Taxonomy

Another Taxonomy



Use Cases

Motivations

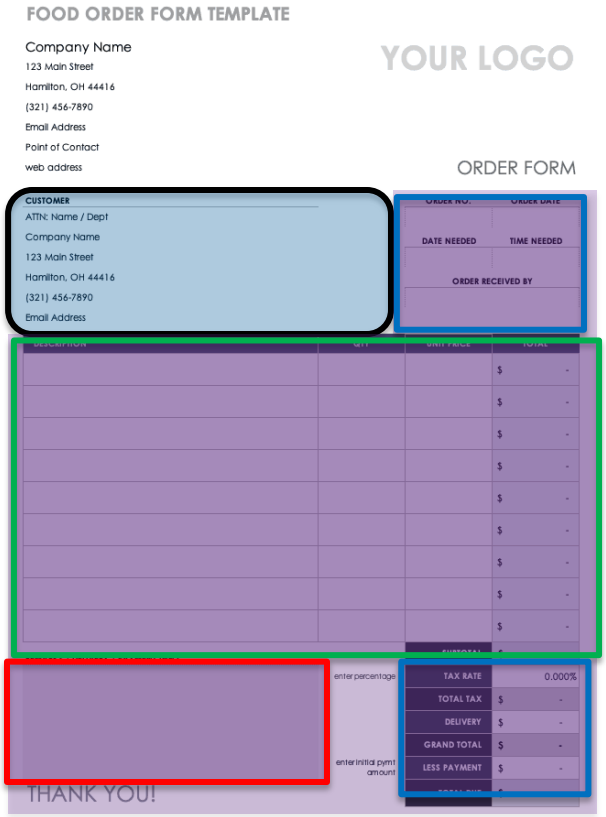
- Massive write performance.
- Fast key value look ups.
- Flexible schema and data types.
- No single point of failure.
- Fast prototyping and development.
- Out of the box scalability.
- Easy maintenance.

What is wrong with SQL/Relational?

- Nothing. One size fits all? Not really.
- Impedance mismatch. –
  Object Relational Mapping doesn't work quite well.
- Rigid schema design.
- Harder to scale.
- Replication.
- Joins across multiple nodes? Hard.
- How does RDMS handle data growth? Hard.
- Need for a DBA.
- Many programmers are already familiar with it.
- Transactions and ACID make development easy.
- Lots of tools to use.

Documents

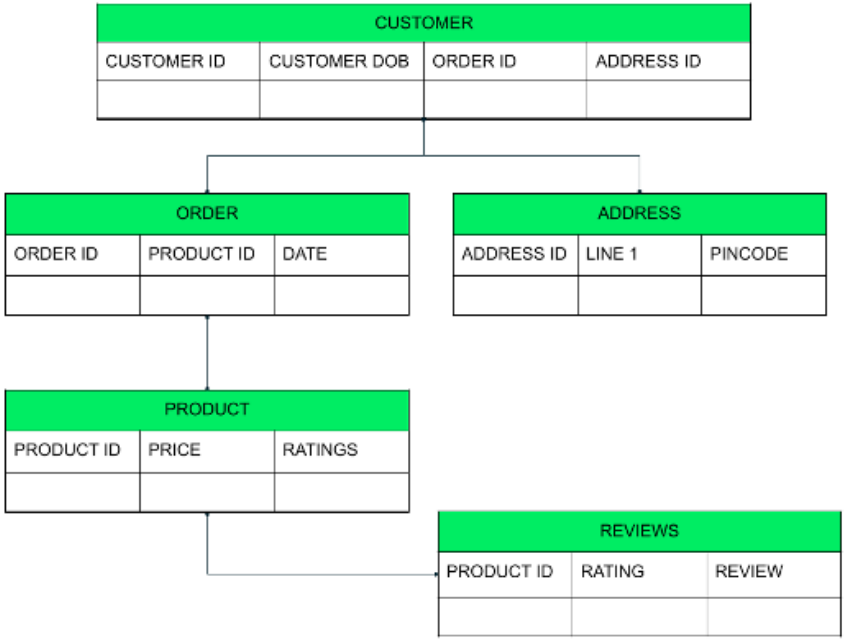
Example Document – An Order



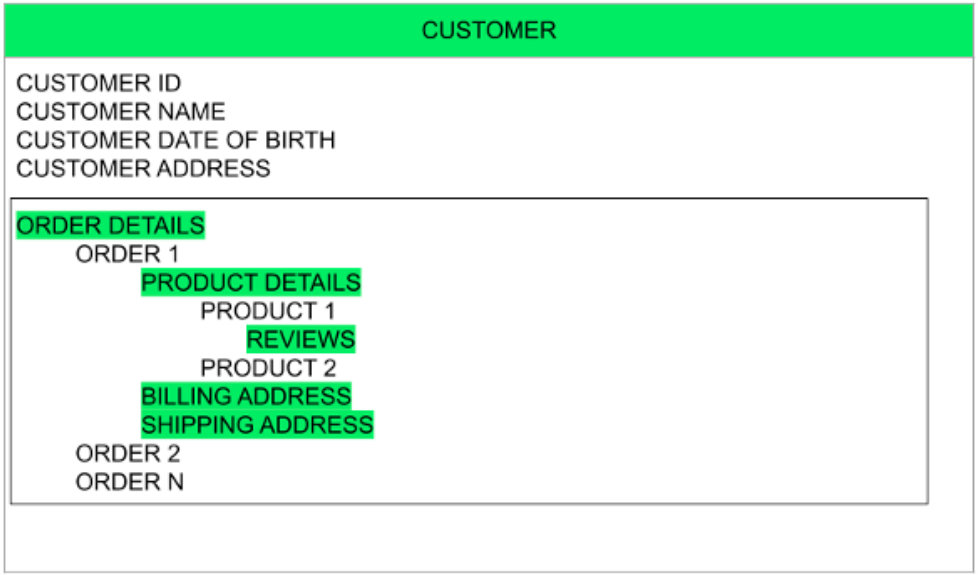
- There are 5 entity types on the form:
  - A "copy of" customer information.
  - Links (via productCode) to products
  - Order
  - OrderDetails
  - Comments
- *But OrderDetails and Comments are somehow different from the others.*
  - *These are arrays of objects*
  - *That are sort of "inside" the order.*
- These are weak entities, but relational does not always handle these well.

Relational vs Document

Relational



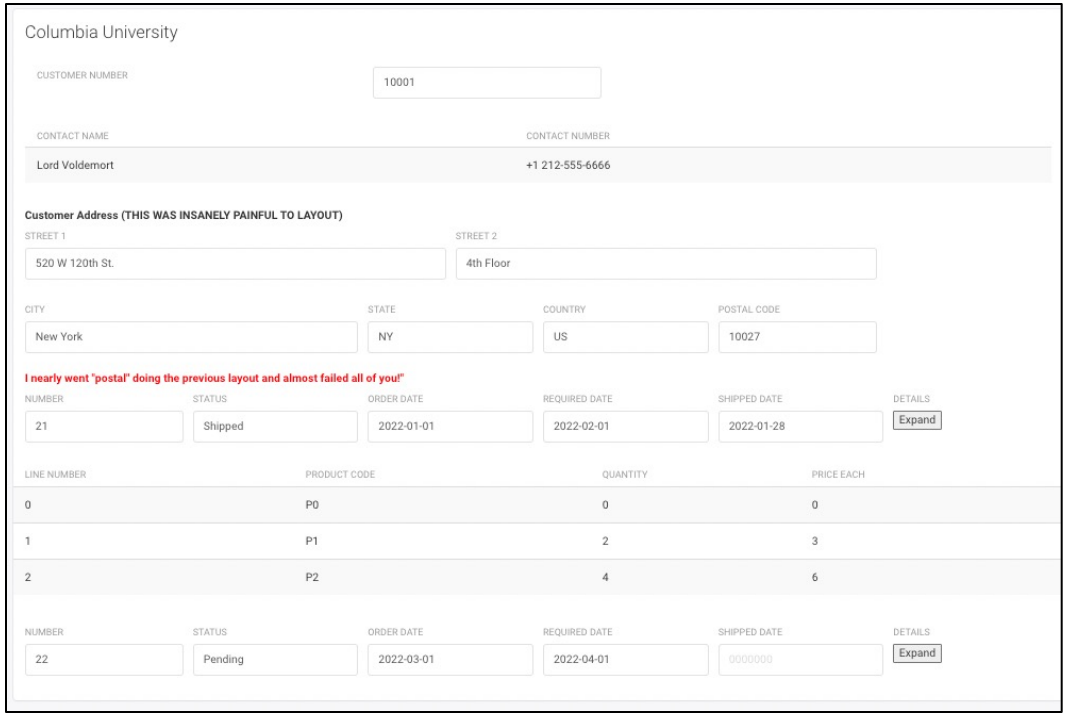
Relational



The key difference?

- Relational DB attributes are atomic.
- Document DB attributes may be *multi-valued* and *complex.*

UI Interface and Logical Data Model




```
{
    customerName: “Columbia University”,
    contact: {
        name: “Lord Voldemort”,
        phone: “+1 212-555-6666”
    },
    address: {
        line1: “520 W 120th St.”
        line2: “Floor 4”,
        … …
    }
    orders: [
        {
            number: …,
            … …
            details: [
                … …
            ]
        }
    ]
}
```

Switch to Notebook

MongoDB

MongoDB Concepts

RDBMS	MongoDB
Database	Database
Table	Collection
Tuple/Row	Document
column	Field
Table Join	Embedded Documents
Primary Key	Primary Key (Default key _id provided by MongoDB itself)
Database Server, Client, Tools, Packages	
mysqld/Oracle	mongod
mysql/sqlplus	mongo
DataGrip	Compass
pymysql	pymongo

Core Operations

Basic Operations:

- Create database
- Create collection
- Create-Retrieve-Update-Delete (CRUD):
  - Create: insert()
  - Retrieve:
    - find()
    - find_one()
  - Update: update()
  - Delete: remove()

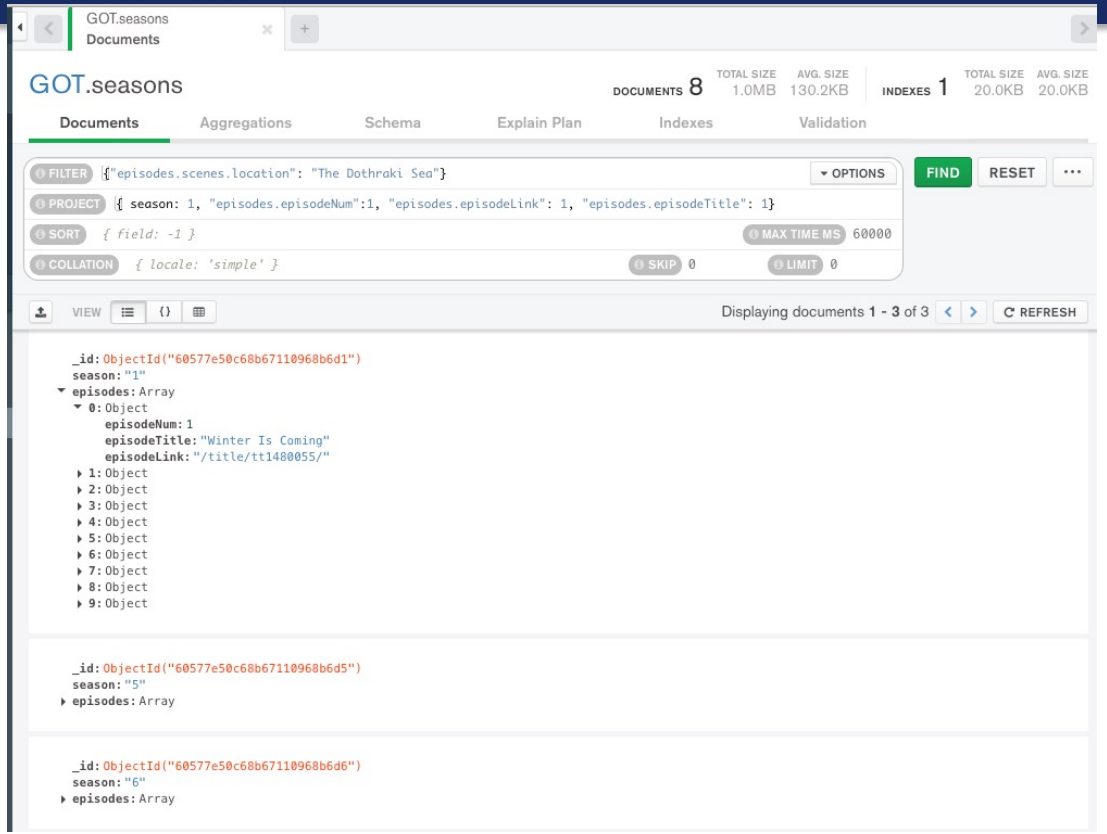
More Advanced Concepts:

- Limit
- Sort
- Aggregation Pipelines
  - Merge
  - Union
  - Lookup
  - Match
  - Merge
  - Sample
  - … …

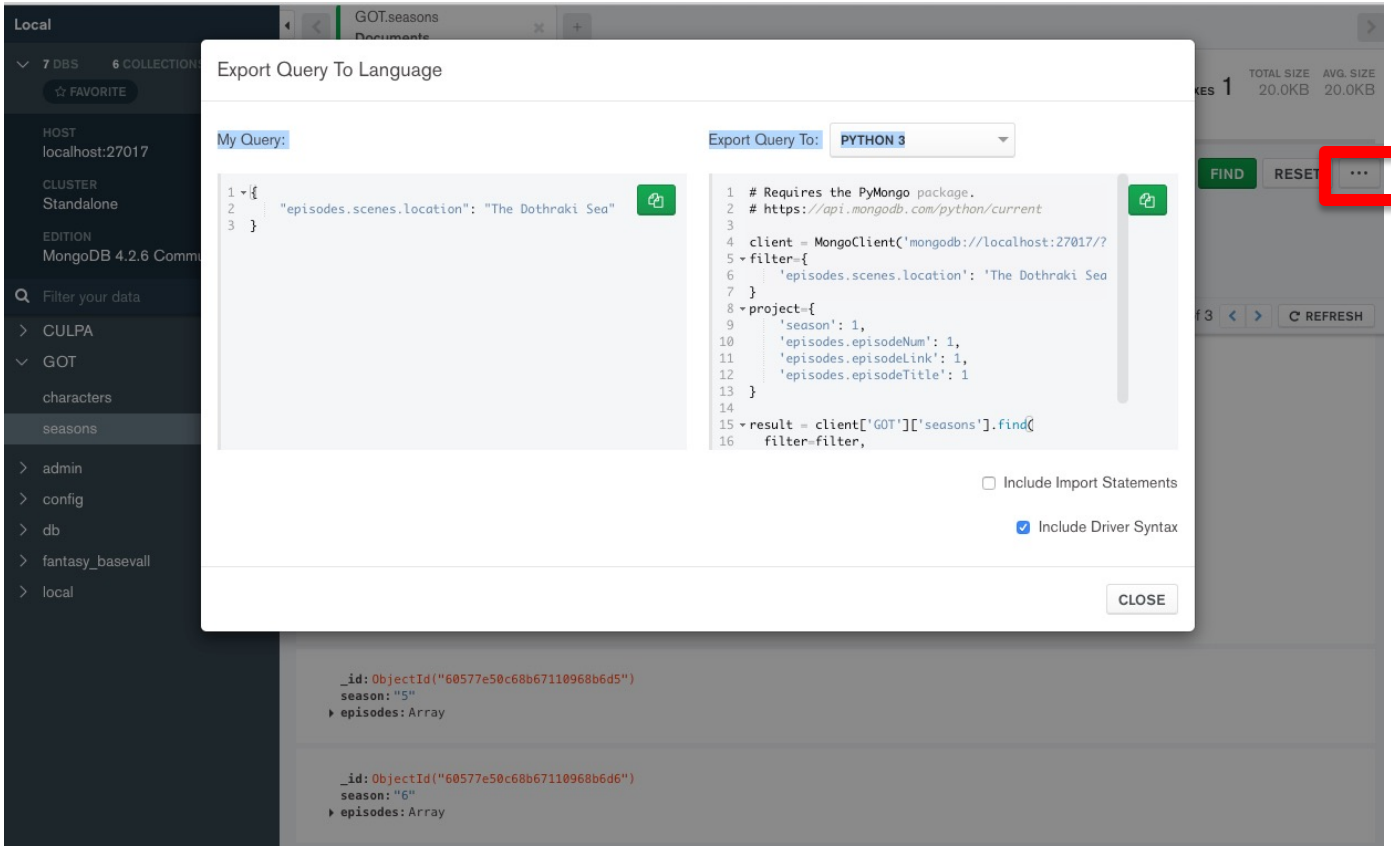
We will just cover the basics for now and may cover more things in HW or other lectures.

find()

- Note:
  - MongoDB uses a more pymysql approach, e.g. an API, than pure declarative languages like SQL.
  - The parameters for find() are where the declarative language appears.
- The basic forms of find() and find_one() have two parameters:
  - *filter expression*
  - *Project expression*
- You can use the Compass tool and screen captures for some HW and exam answers.
- What if I want the answer in a Jupyter Notebook?



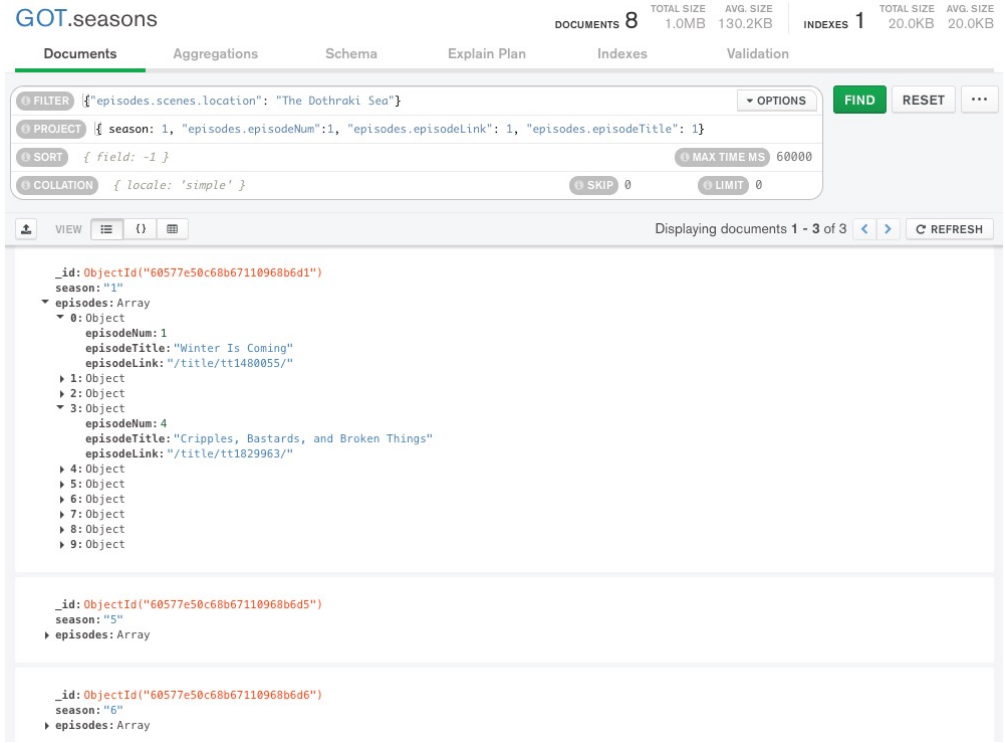
Generate Code



- Choose Export to Language.
- Copy into the notebook.
- The export has an option to include all the connection setup, choosing DB, ...

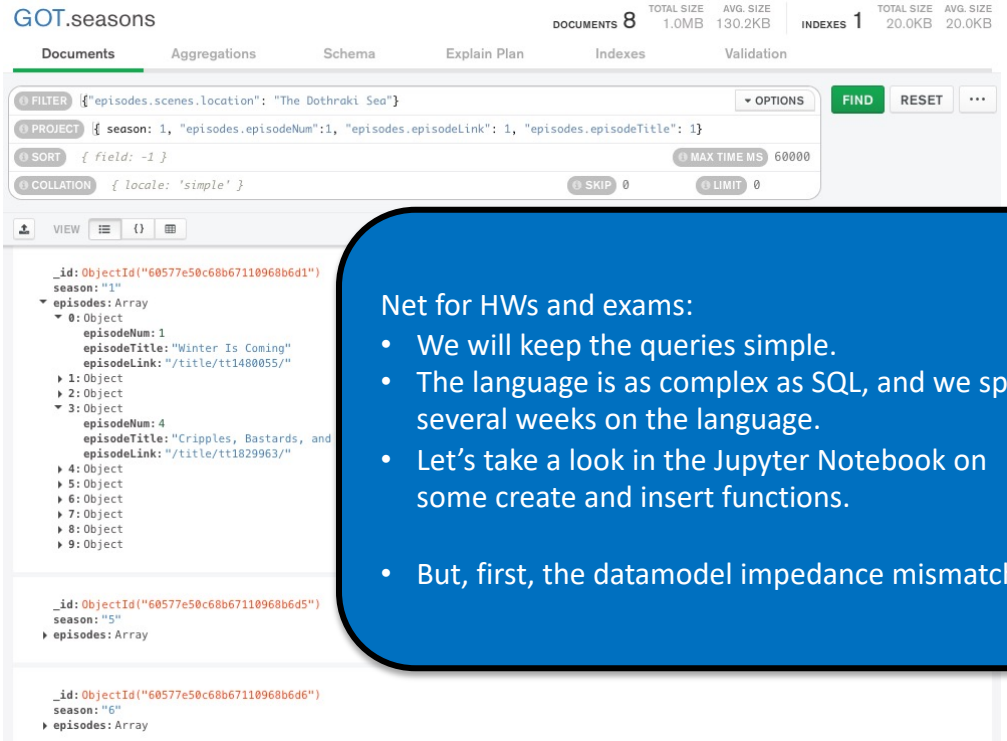
- Switch to Notebook

Result is not Quite You Expect



- The query returns documents that match.
  - The document is “Large” and has and episodes and seasons.
  - If you do a $project requesting episodes/episode content,
    - You get all episodes in the documents that match.
    - Not just the episodes with the scene/location.
    - Projecting array elements from arrays whose elements are arrays is complex and baffling.
- You also get back something (a cursor) that is iterable.

Result is not Quite You Expect



Net for HWs and exams:

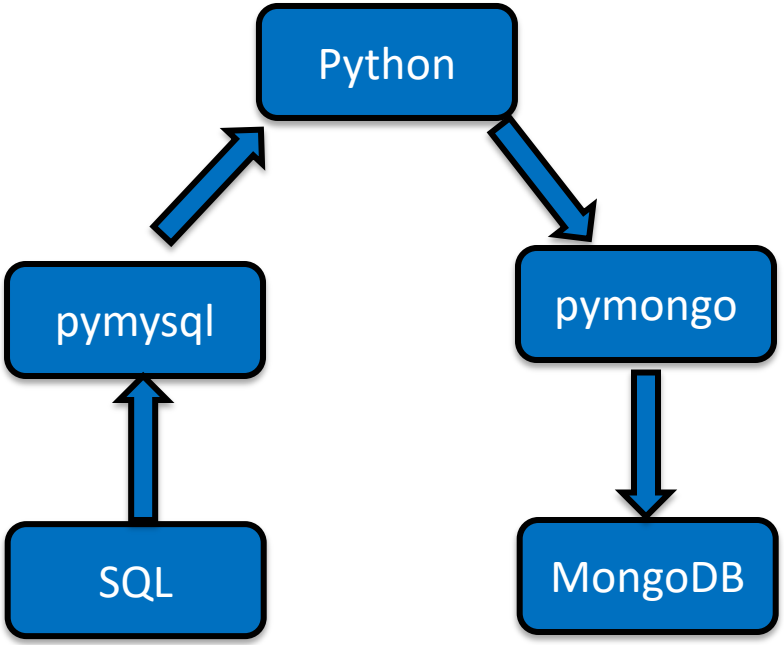
- We will keep the queries simple.
- The language is as complex as SQL, and we spent several weeks on the language.
- Let's take a look in the Jupyter Notebook on some create and insert functions.
- But, first, the datamodel impedance mismatch concept.

- The query returns documents that match.
  - The document is "Large" and has
    episodes and seasons.
    u do a $project requesting
    odes/episode content,
    You get all episodes in the documents that match.
    Not just the episodes with the scene/location.
    Projecting array elements from arrays whose elements are arrays is complex and baffling.
- You also get back something (a cursor) that is iterable.

More Fun – Data Types

MongoDB supports many datatypes. Some of them are −

- **String** − This is the most commonly used datatype to store the data. String in MongoDB must be UTF-8 valid.
- **Integer** − This type is used to store a numerical value. Integer can be 32 bit or 64 bit depending upon your server.
- **Boolean** − This type is used to store a boolean (true/ false) value.
- **Double** − This type is used to store floating point values.
- **Min/ Max keys** − This type is used to compare a value against the lowest and highest BSON elements.
- **Arrays** − This type is used to store arrays or list or multiple values into one key.
- **Timestamp** − ctimestamp. This can be handy for recording when a document has been modified or added.
- **Object** − This datatype is used for embedded documents.
- **Null** − This type is used to store a Null value.
- **Symbol** − This datatype is used identically to a string; however, it's generally reserved for languages that use a specific symbol type.
- **Date** − This datatype is used to store the current date or time in UNIX time format. You can specify your own date time by creating object of Date and passing day, month, year into it.
- **Object ID** − This datatype is used to store the document's ID.
- **Binary data** − This datatype is used to store binary data.
- **Code** − This datatype is used to store JavaScript code into the document.
- **Regular expression** − This datatype is used to store regular expression.



(Some) MongoDB CRUD Operations

- Create:
  - db.collection.insertOne()
  - db.collection.insertMany()
- Retrieve:
  - db.collection.find()
  - db.collection.findOne()
  - db.collection.findOneAndUpdate()
  - … ..
- Update:
  - db.collection.updateOne()
  - db.collection.updateMany()
  - db.collection.replaceOne()
- Delete:
  - db.collection.deleteOne()
  - db.collection.deleteMany()

pymongo maps the camel case to _, e.g.

- findOne()
- find_one()

There are good online tutorials:

- https://www.tutorialspoint.com/python_data_access
- https://www.tutorialspoint.com/mongodb/index.htm

(Some) MongdoDB Pipeline Operators

https://www.slideshare.net/mongodb/s01-e04-analytics

Aggregation operators

- Pipeline and Expression operators

Pipeline	Expression	Arithmetic	Conditional
$match	$addToSet	$add	$cond
$sort	$first	$divide	$ifNull
$limit	$last	$mod	
$skip	$max	$multiply	
$project	$min	$subtract	
$unwind	$avg		Variables
$group	$push		
$geoNear	$sum		$let
$text			$map
$search			

Tip: Other operators for date, time, boolean and string manipulation

MongoDB Checkpoint

- You can see that MongoDB has powerful, sophisticated
  - Operators
  - Expressions
  - Pipelines
- We have only skimmed the surface. There is a lot more:
  - Indexes
  - Replication, Sharding
  - Embedded Map-Reduce support
  - … …
- We will explore a little more in subsequent lectures, homework, … …
- You will have to install MongoDB and Compass for HW4 and final exam.