W4111 – Introduction to Databases
Section 002, Fall 2021

Lecture 8: No SQL Intro., Module II Intro

COLUMBIA | ENGINEERING
The Fu Foundation School of Engineering and Applied Science

Contents

Course Modules – Reminder

Course Overview

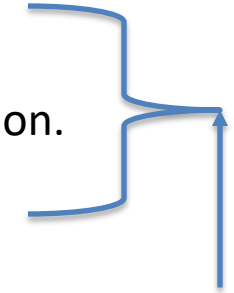
Each section of W4111 is slightly different based on student interest and professor's focus. There is a common, core syllabus. Professors cover topics in different orders and grouping based on teaching style.

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.

Today's Contents

- Agenda update – Parallel coverage of
  - Module II: DBMS internal architecture and implementation.
  - Module III: NoSQL.
- Module II:
  - Overview.
  - Major subsystems summary.
  - Database disks and files.
- Module III:
  - Overview and NoSQL concepts.
  - Graph databases and Neo4j.
- HW 3/Project specification and discussion.



I am going to cover module II in chunks over multiple lectures:

- Material is on the *required* syllabus.
- Interesting and fascinating.
- But, No SQL is more broadly applicable to how students will use databases.

Module II Kickoff

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 induplication 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

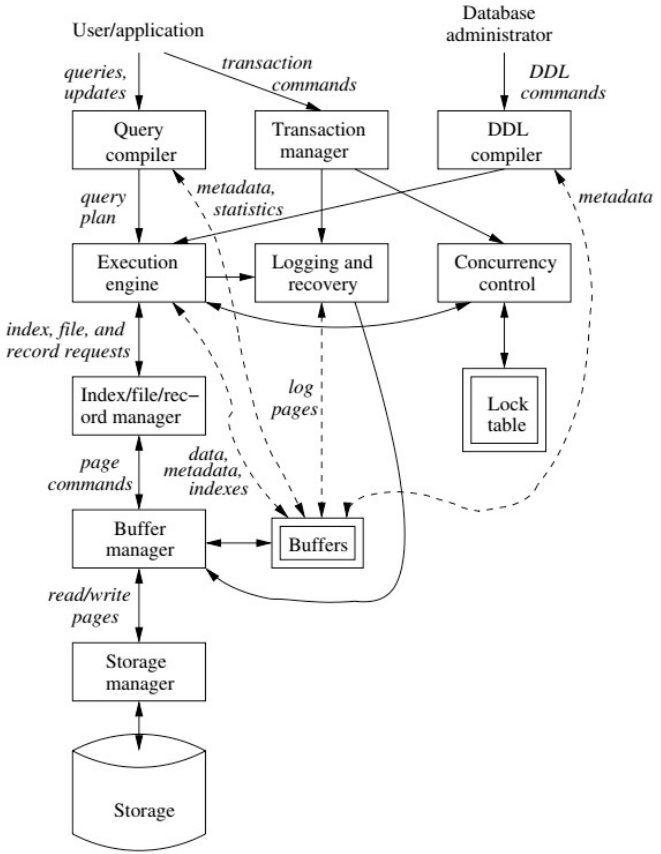
Purpose of Database Systems (Cont.)

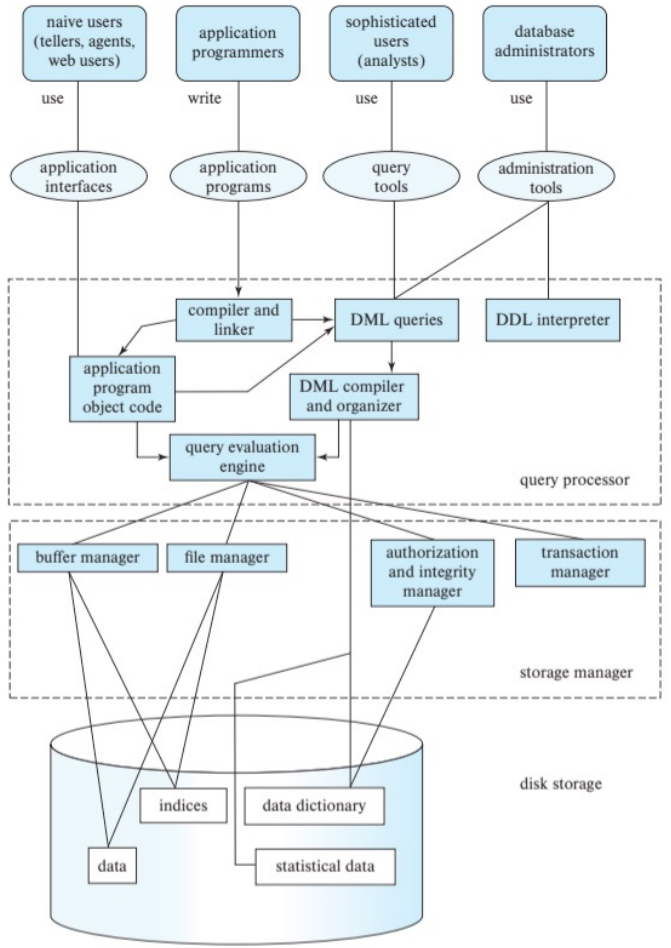
- Atomicity of updates
  - Failures may leave database in an inconsistent state with partial updates carried out
  - Example: Transfer of funds from one account to another should either complete or not happen at all
- Concurrent access by multiple users
  - Concurrent access needed for performance
  - 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."

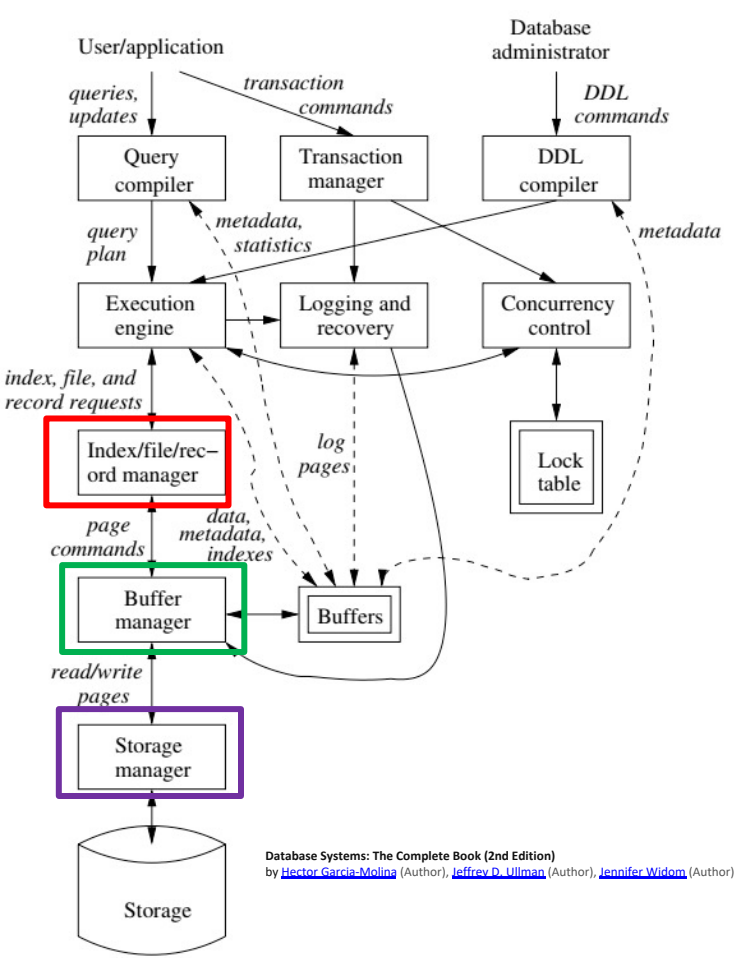
DBMS Arch.





Data Management

- Find things quickly.
- Access things quickly.
- Load/save things quickly.



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

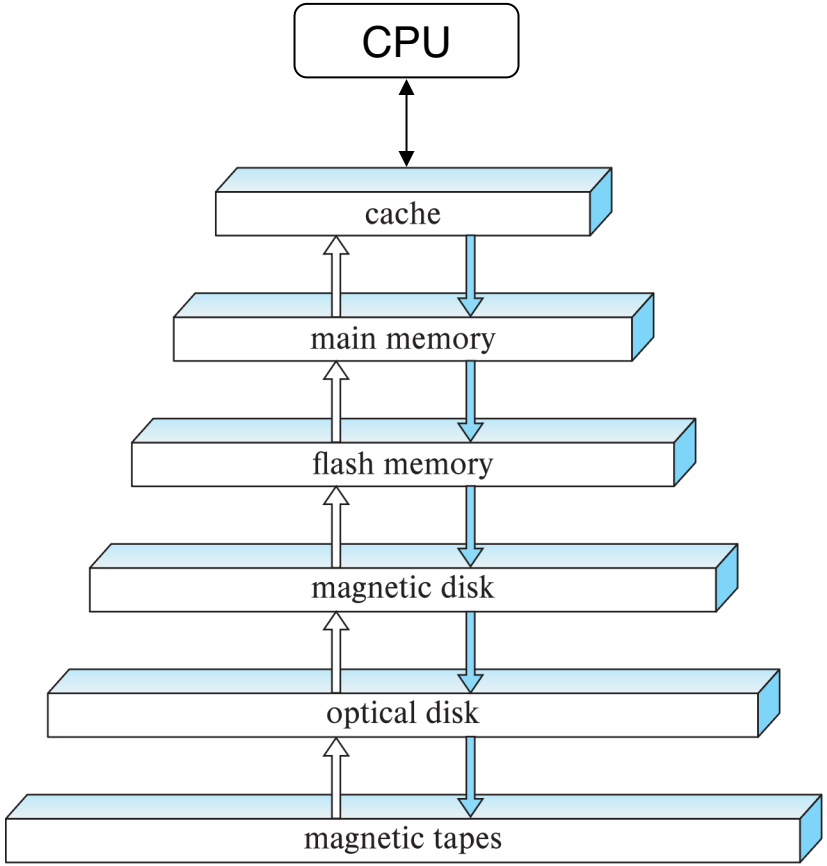
Disks
Input/Output (IO)

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

Storage Hierarchy



Memory Hierarchy

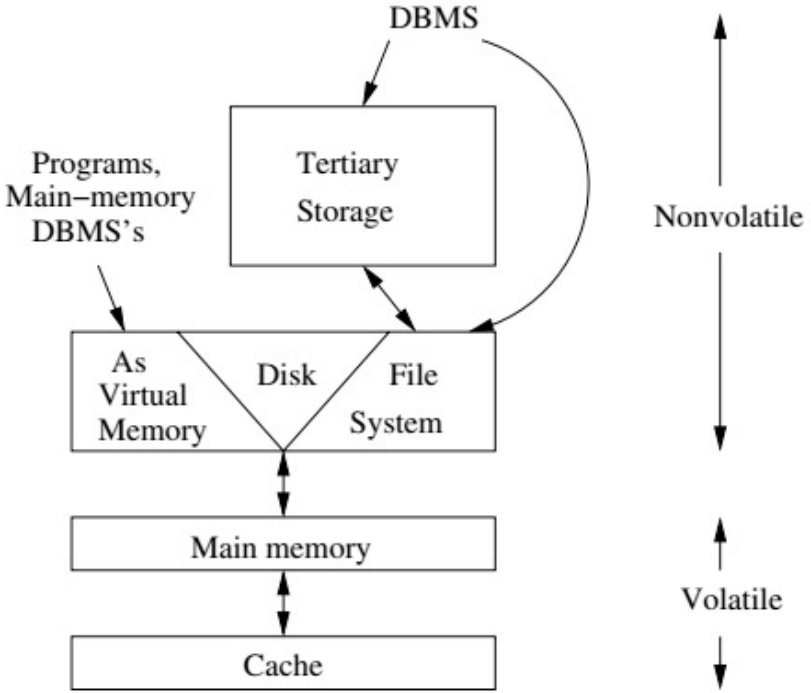


Figure 13.1: The memory hierarchy

Slower

Cheaper

Unlimited

Faster

More expensive

Limited

Memory Hierarchy (Very Old Numbers – Still Directionally Valid)

Storage Technology

Price, Performance & Capacity

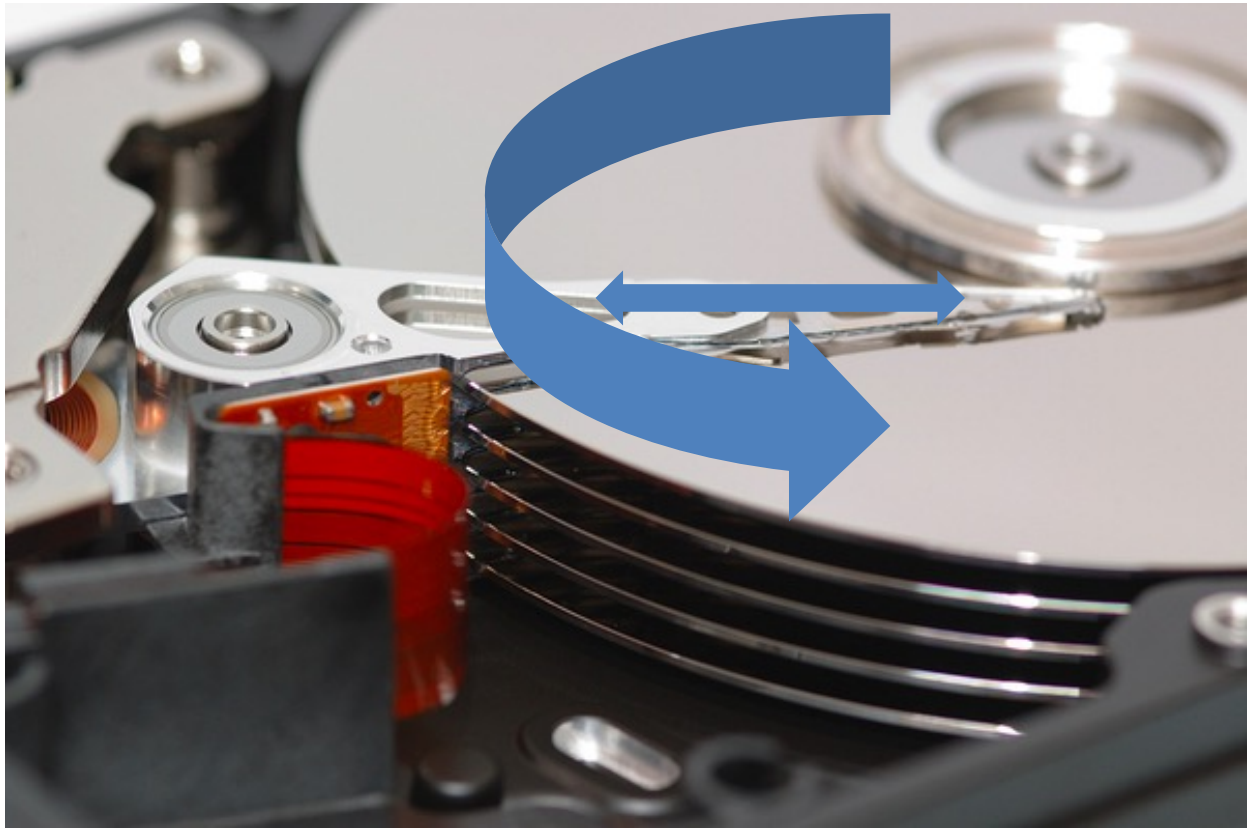
Technologies	Capacity (GB)	Latency (microS)	IOPs	Cost/IOPS ($)	Cost/GB ($)
Cloud Storage	*Unlimited*	*60,000*	*20*	*17c/GB*	*0.15/month*
Capacity HDDs	*2,500*	*12,000*	*250*	*1.67*	*0.15*
Performance HDDs	*300*	*7,000*	*500*	*1.52*	*1.30*
SSDs (write)	*64*	*300*	*5000*	*0.20*	*13*
SSDs (read only)	*64*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

Hard Disk Drive



Disk Configuration

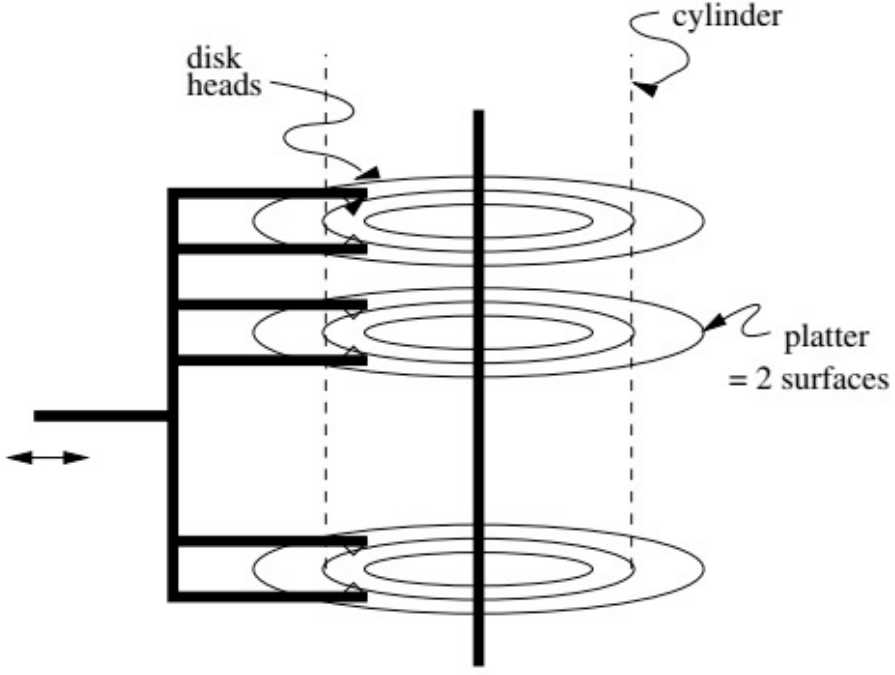


Figure 13.2: A typical disk

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.

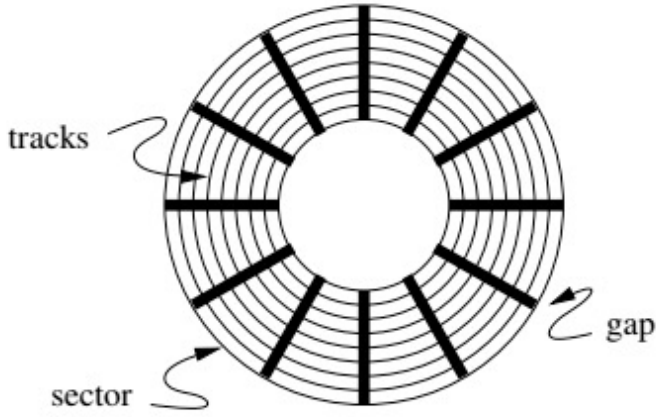
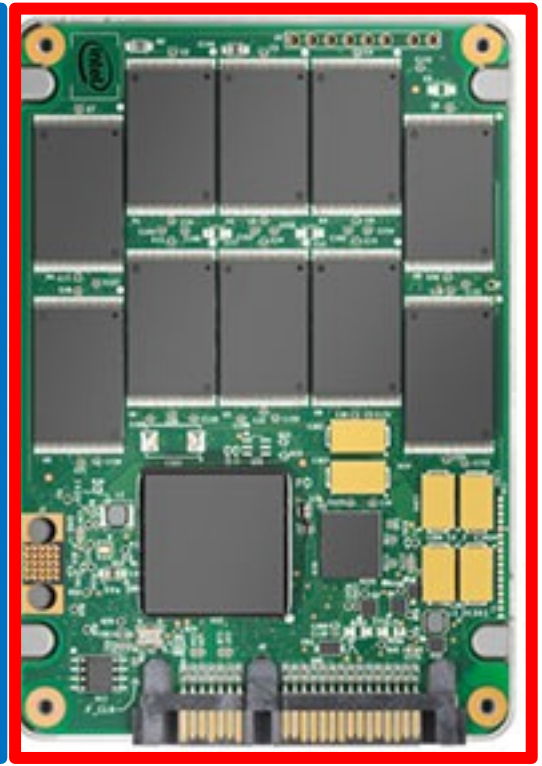


Figure 13.3: Top view of a disk surface

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

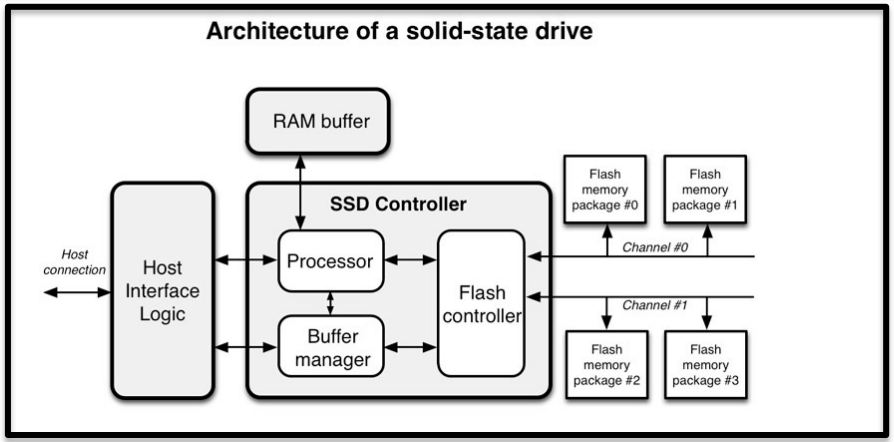
- NOR flash vs NAND flash
- NAND flash
  - used widely for storage, cheaper than NOR flash
  - requires page-at-a-time read (page: 512 bytes to 4 KB)
    - 20 to 100 microseconds for a page read
    - Not much difference between sequential and random read
  - Page can only be written once
    - Must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

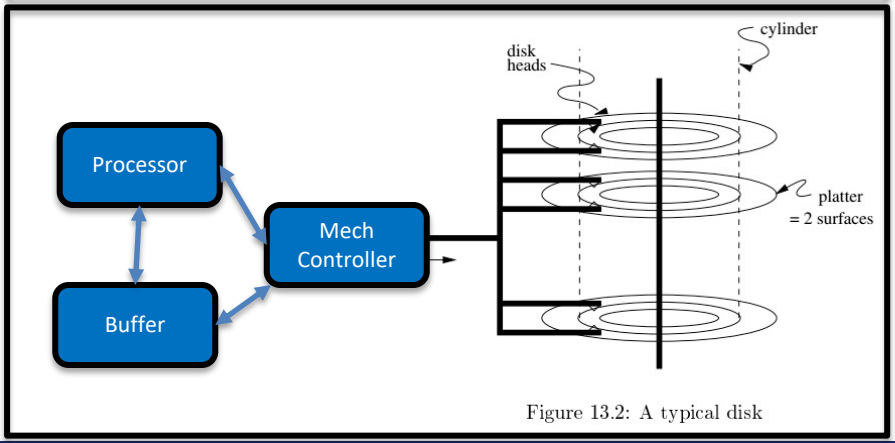


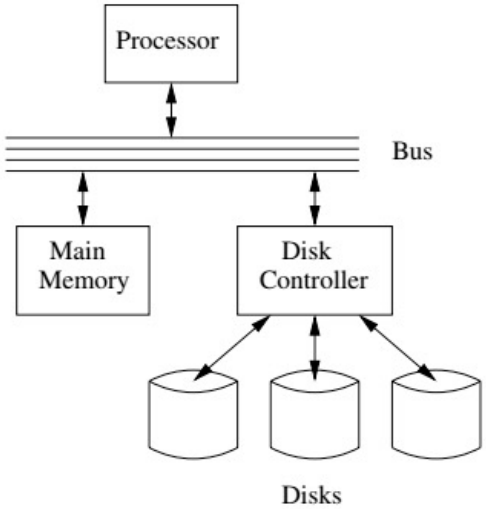
Figure 13.2: A typical disk

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

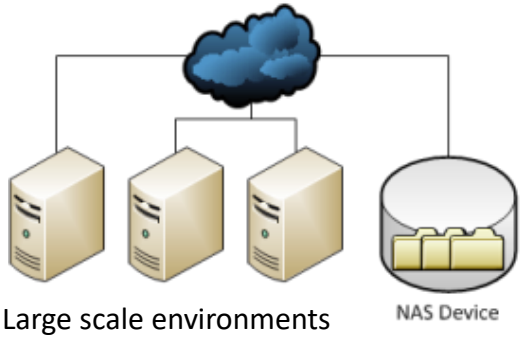
I/O Architecture



How we normally think of disks and I/O.

Network Attached Storage

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

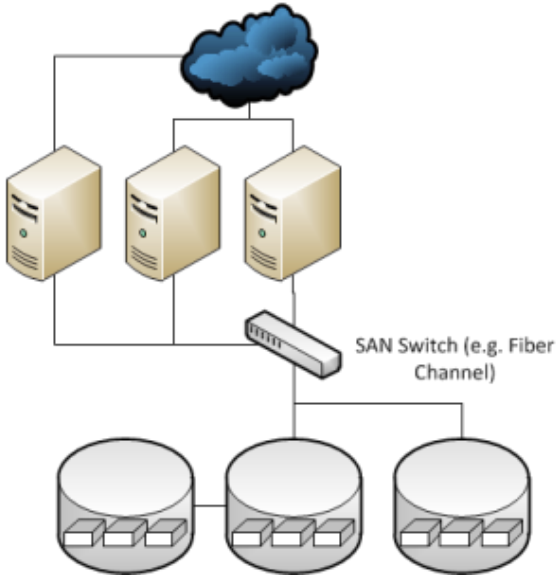


Large scale environments

- 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

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**

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

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

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

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. (…)

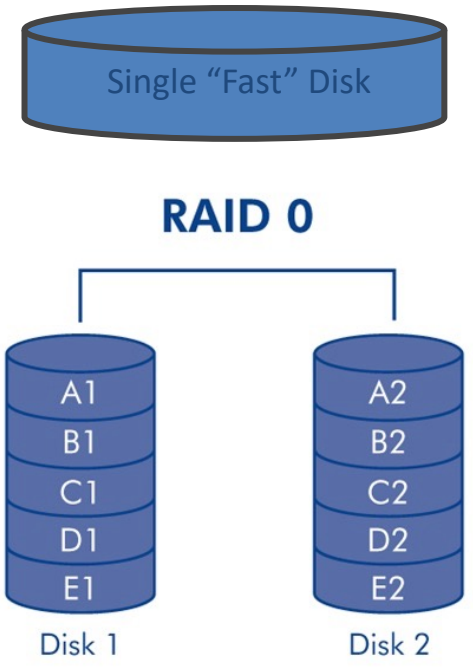
RAID 0 consists of striping, without mirroring or parity. (…)
RAID 1 consists of data mirroring, without parity or striping. (…)
RAID 2 consists of bit-level striping with dedicated Hamming-code parity. (…)
RAID 3 consists of byte-level striping with dedicated parity. (…)
RAID 4 consists of block-level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

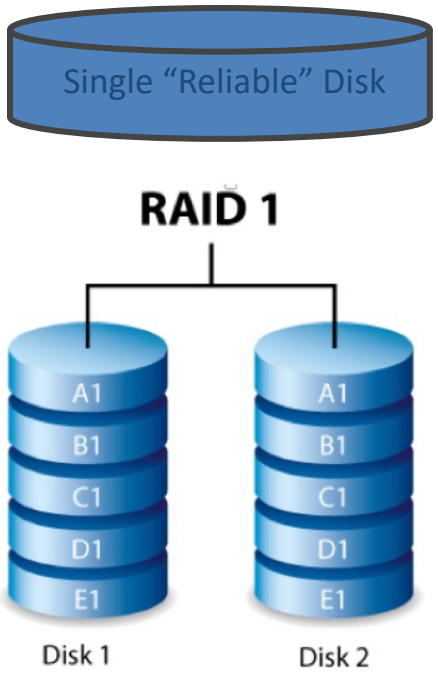
- RAID 0+1: creates two stripes and mirrors them. (…)
- RAID 1+0: creates a striped set from a series of mirrored drives. (…)
- **JBOD RAID N+N:** With JBOD (*just a bunch of disks*), (…)"

RAID-0 and RAID-1

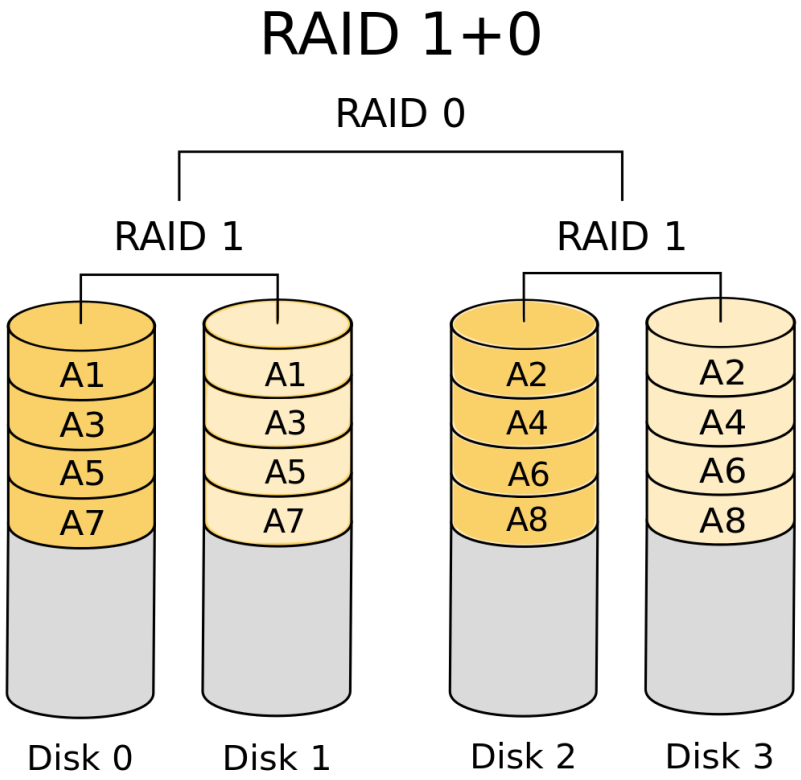
Two physical disks make one single, logical **fast** disk



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



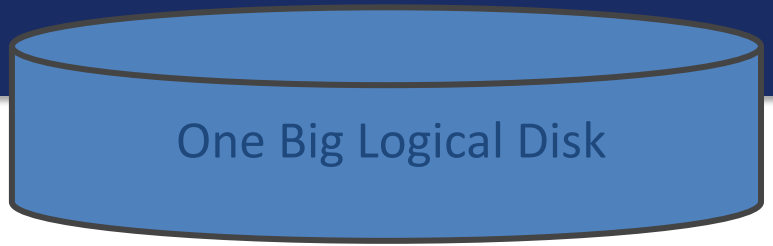
Mixed RAID Modes



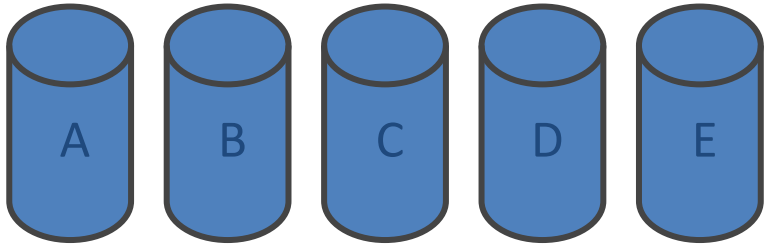
Stripe
And
Mirror

RAID-5

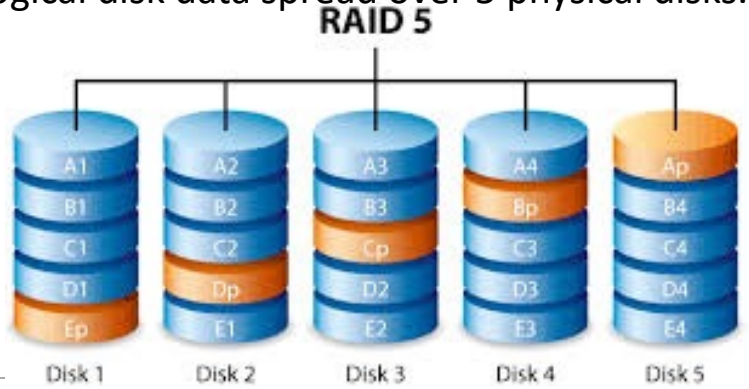
- Improved performance through parallelism
  - Rotation/seek
  - Transfer
- Availability uses *parity blocks*
  - Suppose I have 4 different data blocks on the logical drive A: A1, A2, A3, A4.
  - Parity function: **Ap** = P(A1,A2,A3,A4)
  - Recovery function: A2=R(**Ap**,A1,A3,A4)
- During normal operations:
  - 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

  .

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

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

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

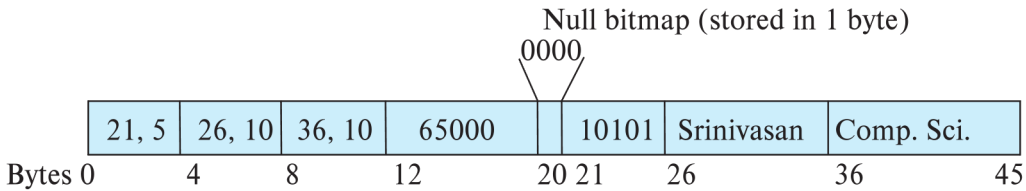
Fixed-Length Records

- 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***

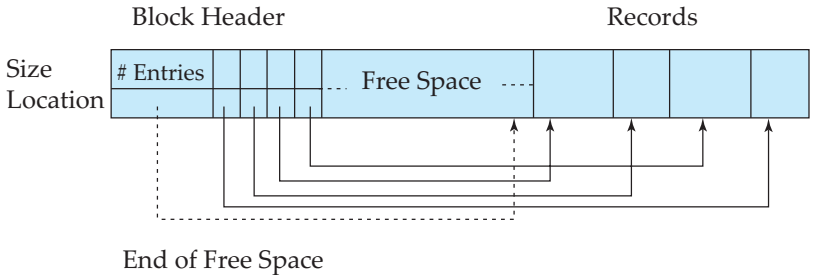
header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology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

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



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.

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

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

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)

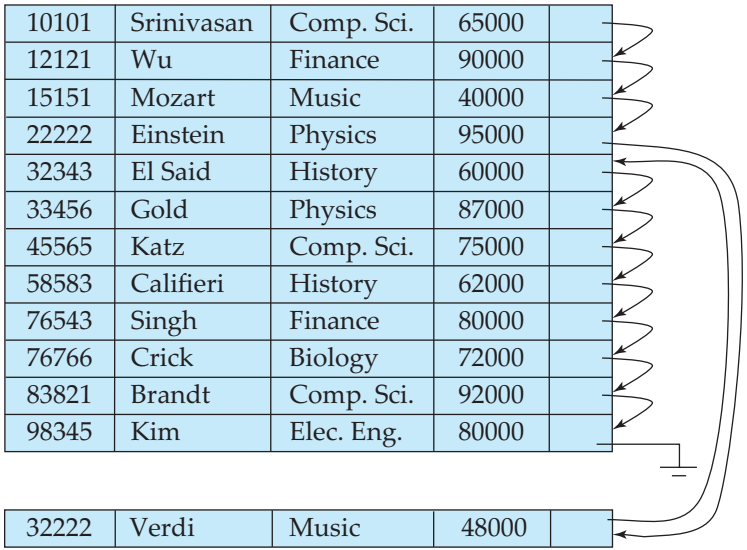
Sequential File Organization

- 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	

Sequential File Organization (Cont.)

- 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

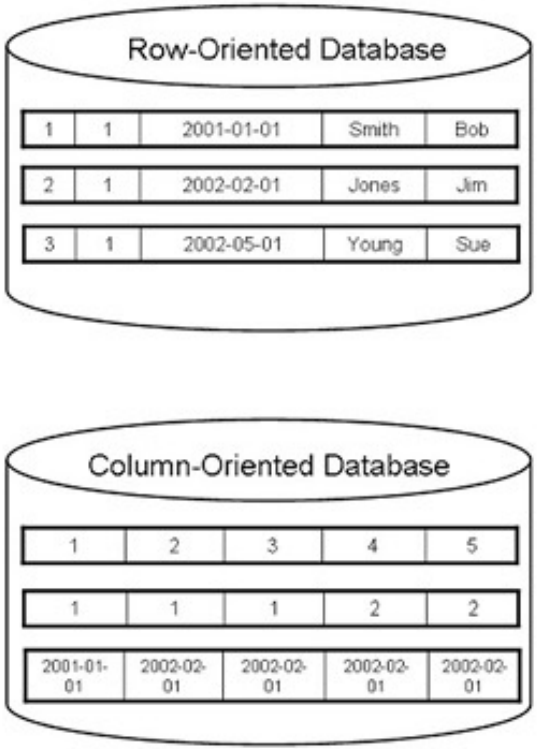
Columnar Representation

- 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**

Row vs Column

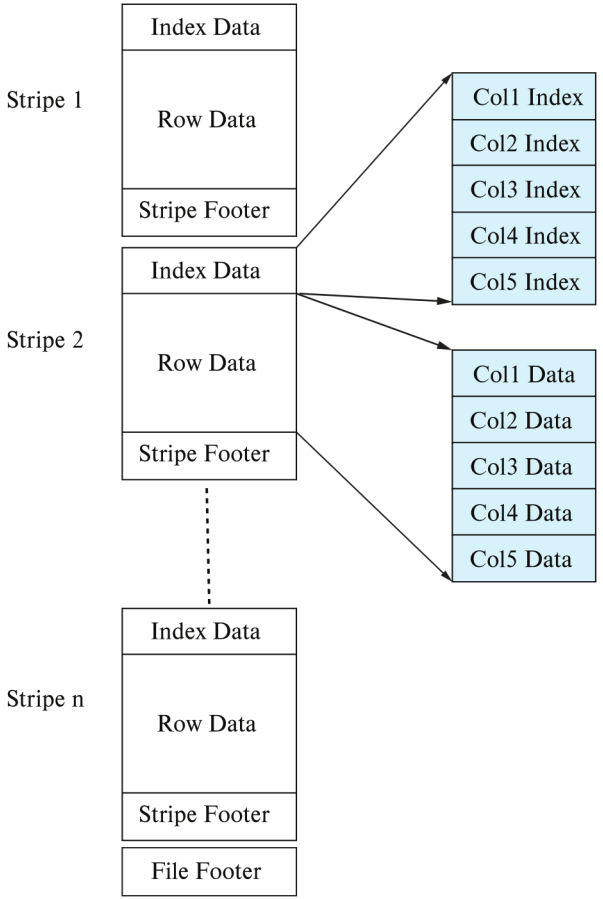
- Columnar and Row are both
  - Relational
  - Support SQL operations
- But differ in data storage
  - Row keeps row data together in blocks.
  - Columnar keys column data together in blocks.
- This determines performance for diffe types of query, e.g.
  - Columnar is extremely powerful for B
    - Aggregation ops, e.g. SUM, AVG
    - PROJECT (do not load all of the row) t
  - 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

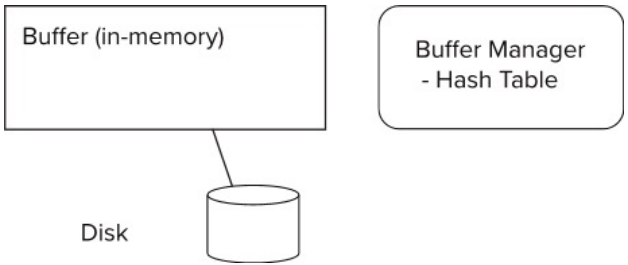
- ORC and Parquet: file formats with columnar storage inside file
- Very popular for big-data applications
- Orc file format shown on right:



Memory and Buffer Pools

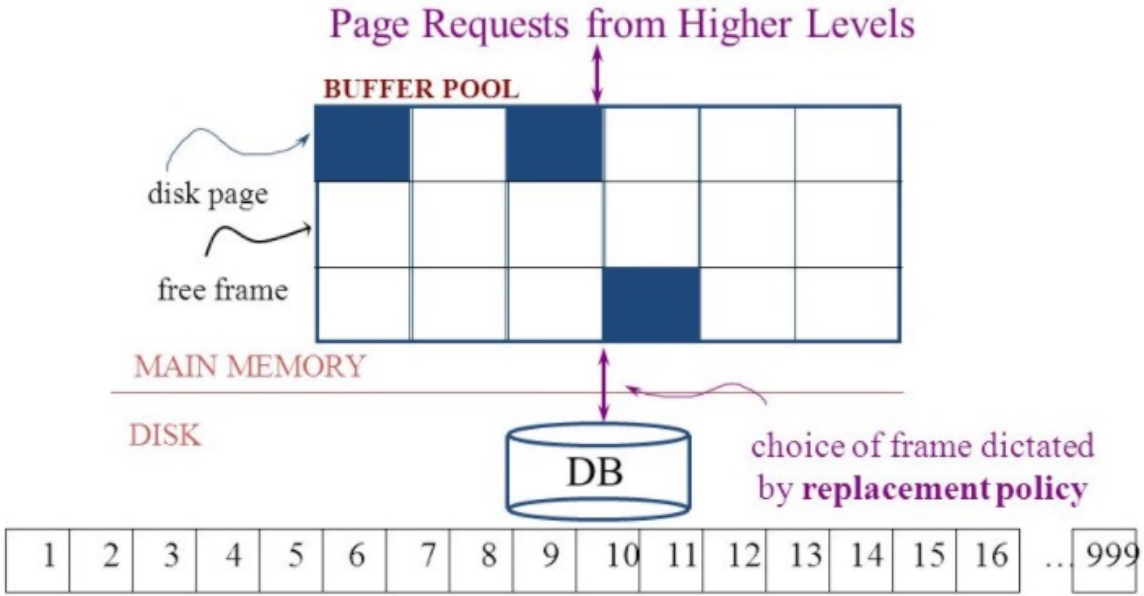
Storage Access

- Blocks are units of both storage allocation and data transfer.
- Database system seeks to minimize the number of block transfers between the disk and memory. We can reduce the number of disk accesses by keeping as many blocks as possible in main memory.
- **Buffer** – portion of main memory available to store copies of disk blocks.
- **Buffer manager** – subsystem responsible for allocating buffer space in main memory.



The Logical Concept

- The DBMS and queries can only manipulate in-memory blocks and records.
- A very, very, very small fraction of all blocks fit in memory.



Buffer Manager

- Programs call on the buffer manager when they need a block from disk.
  - If the block is already in the buffer, buffer manager returns the address of the block in main memory
  - If the block is not in the buffer, the buffer manager
    - Allocates space in the buffer for the block
      - Replacing (throwing out) some other block, if required, to make space for the new block.
      - Replaced block written back to disk only if it was modified since the most recent time that it was written to/fetched from the disk.
    - Reads the block from the disk to the buffer, and returns the address of the block in main memory to requester.

Buffer Manager

- **Buffer replacement strategy** (details coming up!)
- **Pinned block:** memory block that is not allowed to be written back to disk
  - **Pin** done before reading/writing data from a block
  - **Unpin** done when read /write is complete
  - Multiple concurrent pin/unpin operations possible
    - Keep a pin count, buffer block can be evicted only if pin count = 0
- **Shared and exclusive locks on buffer**
  - Needed to prevent concurrent operations from reading page contents as they are moved/reorganized, and to ensure only one move/reorganize at a time
  - Readers get shared lock, updates to a block require exclusive lock
  - **Locking rules:**
    - Only one process can get exclusive lock at a time
    - Shared lock cannot be concurrently with exclusive lock
    - Multiple processes may be given shared lock concurrently

Buffer-Replacement Policies

- Most operating systems replace the block **least recently used** (LRU strategy)
  - Idea behind LRU – use past pattern of block references as a predictor of future references
  - LRU can be bad for some queries
- Queries have well-defined access patterns (such as sequential scans), and a database system can use the information in a user's query to predict future references
- Mixed strategy with hints on replacement strategy provided by the query optimizer is preferable
- Example of bad access pattern for LRU: when computing the join of 2 relations r and s by a nested loops

```
for each tuple tr of r do
  for each tuple ts of s do
    if the tuples tr and ts match …
```

Buffer-Replacement Policies (Cont.)

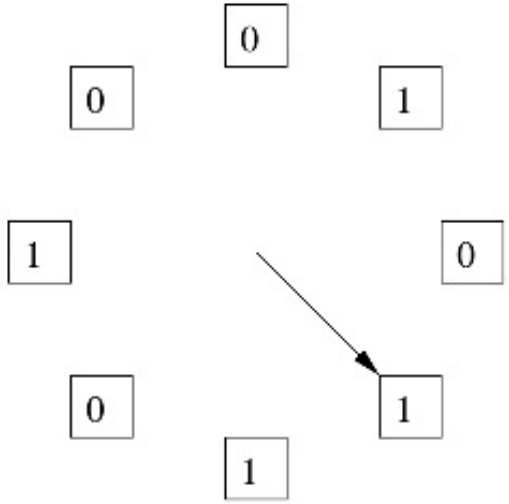
- **Toss-immediate** strategy – frees the space occupied by a block as soon as the final tuple of that block has been processed
- **Most recently used (MRU) strategy** – system must pin the block currently being processed. After the final tuple of that block has been processed, the block is unpinned, and it becomes the most recently used block.
- Buffer manager can use statistical information regarding the probability that a request will reference a particular relation
  - E.g., the data dictionary is frequently accessed. Heuristic: keep data-dictionary blocks in main memory buffer
- Operating system or buffer manager may reorder writes
  - Can lead to corruption of data structures on disk
    - E.g., linked list of blocks with missing block on disk
    - File systems perform consistency check to detect such situations
  - Careful ordering of writes can avoid many such problems

Replacement Policy

- The *replacement policy* is one of the most important factors in database management system implementation and configuration.
- A very simple, introductory explanation is (https://en.wikipedia.org/wiki/Cache_replacement_policies).
  - There are a lot of possible policies.
  - The *most* efficient caching algorithm would be to always discard the information that will not be needed for the longest time in the future. This optimal result is referred to as Bélády's optimal algorithm/simply optimal replacement policy or the clairvoyant algorithm.
- All implementable policies are an attempt to approximate knowledge of the future based on knowledge of the past.
- Least Recently Used is based on the simplest assumption
  - The information that will not be needed for the longest time.
  - Is the information that has not been accessed for the longest time.

The “Clock Algorithm”

- LRU is (perceived to be) expensive
  - Maintain timestamp for each block.
  - Update and resort blocks on access.
- The “Clock Algorithm” is a less expensive approximation.
  - Arrange the frames (places blocks can go) into a logical circle like the seconds on a clock face.
  - Each frame is marked 0 or 1.
    - Set to 1 when block added to frame.
    - Or when application accesses a block in frame.
  - Replacement choice
    - Sweep second hand clockwise one frame at a time.
    - If bit is 0, choose for replacement.
    - If bit is 1, set bit to zero and go to next frame.
- The basic idea is. On a clock face
  - If the second hand is currently at 27 seconds.
  - The 28 second tick mark is “the least recently touched mark.”



Replacement Algorithm

The algorithms are more sophisticated in the real world, e.g.

- "Scans" are common, e.g. go through a large query result in order (will be more clear when discussing cursors).
  - The engine knows the current position in the result set.
  - Uses the sort order to determine which records will be accessed soon.
  - Tags those blocks as not replaceable.
  - (A form of clairvoyance).
- Not all users/applications are equally "important."
  - Classify users/applications into priority 1, 2 and 3.
  - Sub-allocate the buffer pool into pools P1, P2 and P3.
  - Apply LRU within pools and adjust pool sizes based on relative importance.
  - This prevents
    - A high access rate, low-priority application from taking up a lot of frames
    - Result in low access, high priority applications not getting buffer hits.

Optimization of Disk Block Access (Cont.)

- Buffer managers support **forced output** of blocks for the purpose of recovery (more in Chapter 19)
- **Nonvolatile write buffers** speed up disk writes by writing blocks to a non-volatile RAM or flash buffer immediately
  - *Writes can be reordered to minimize disk arm movement*
- **Log disk** – a disk devoted to writing a sequential log of block updates
  - Used exactly like nonvolatile RAM
    - Write to log disk is very fast since no seeks are required
- **Journaling file systems** write data in-order to NV-RAM or log disk
  - Reordering without journaling: risk of corruption of file system data

NoSQL

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]

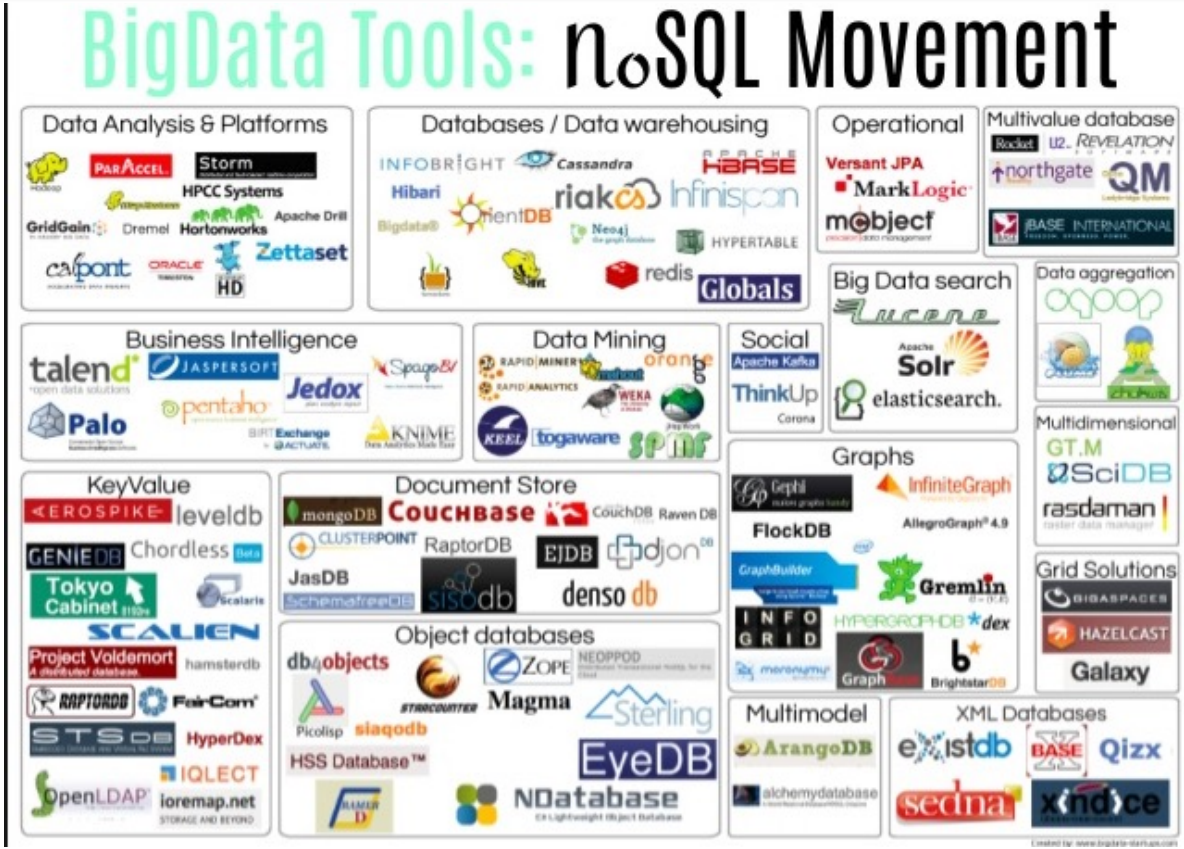
Overview (I) (https://en.wikipedia.org/wiki/NoSQL)

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]

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.

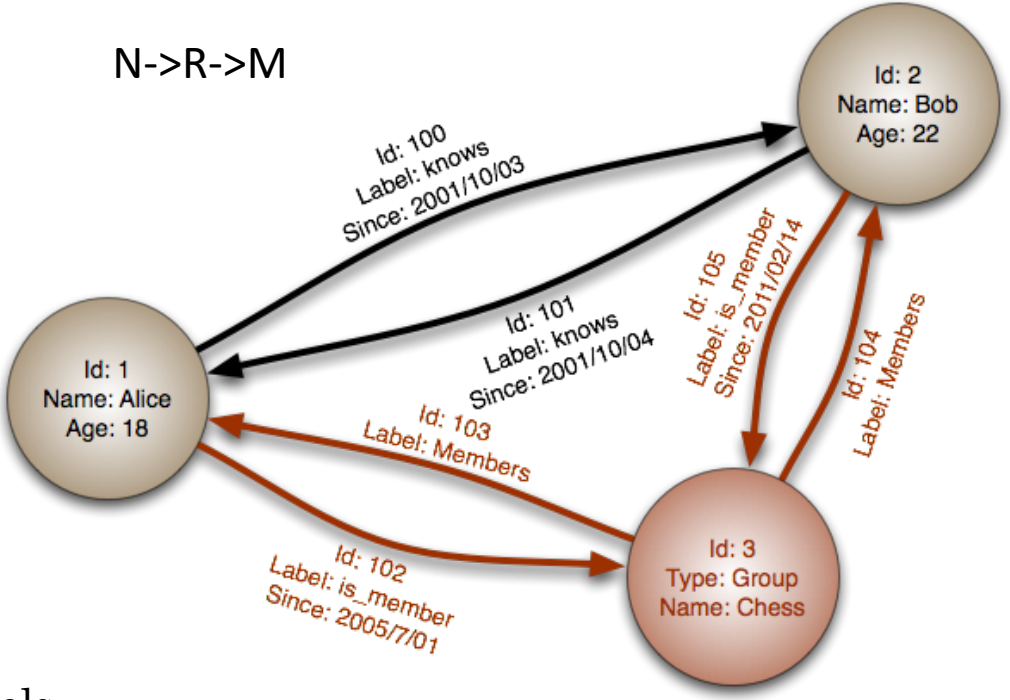
What is wrong with SQL/Relational?

- 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.

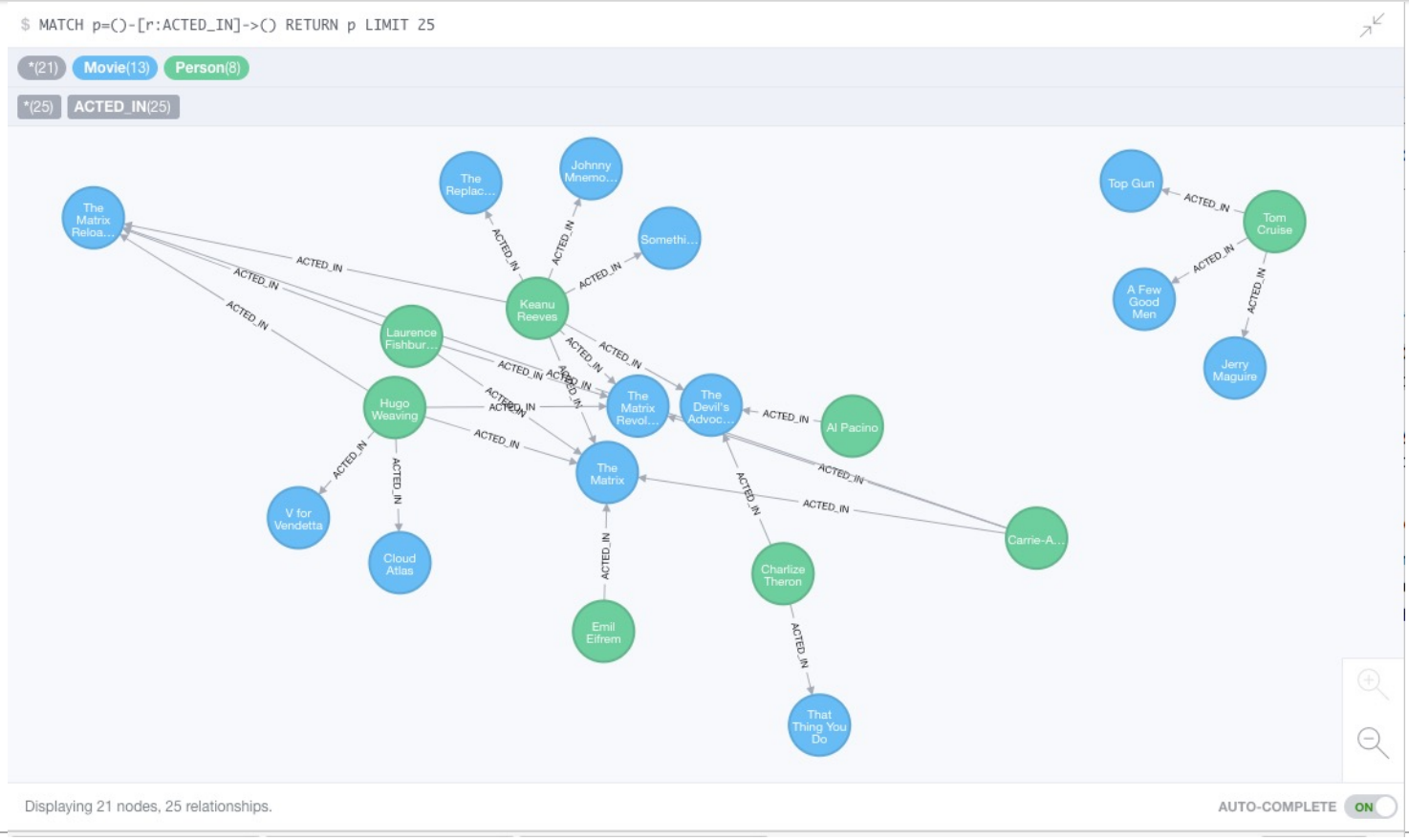
Graph Databases

Graph Database

- Exactly what it sounds like
- Two core types
  - Node
  - Edge (link)
- Nodes and Edges have
  - Label(s) = “Kind”
  - Properties (free form)
- Query is of the form
  - p1(n)-p2(e)-p3(m)
  - n, m are nodes; e is an edge
  - p1, p2, p3 are predicates on labels



Neo4J Graph Query



Why Graph Databases?

- Schema Less and Efficient storage of Semi Structured Information
- No O/R mismatch – very natural to map a graph to an Object Oriented language like Ruby.
- Express Queries as Traversals. Fast deep traversal instead of slow SQL queries that span many table joins.
- Very natural to express graph related problem with traversals (recommendation engine, find shortest parth etc..)
- Seamless integration with various existing programming languages.
- ACID Transaction with rollbacks support.
- Whiteboard friendly – you use the language of node,properties and relationship to describe your domain (instead of e.g. UML) and there is no need to have a complicated O/R mapping tool to implement it in your database. You can say that Neo4j is "Whiteboard friendly" !(http://video.neo4j.org/JHU6F/live-graph-session-how-allison-knows-james/)

Social Network "path exists" Performance

- Experiment:
  - ~1k persons
  - Average 50 friends per person
  - `pathExists(a,b)` limited to depth 4

	# persons	query time
Relational database	1000	2000ms
Neo4j	1000	2ms
Neo4j	1000000	2ms

Graph databases are

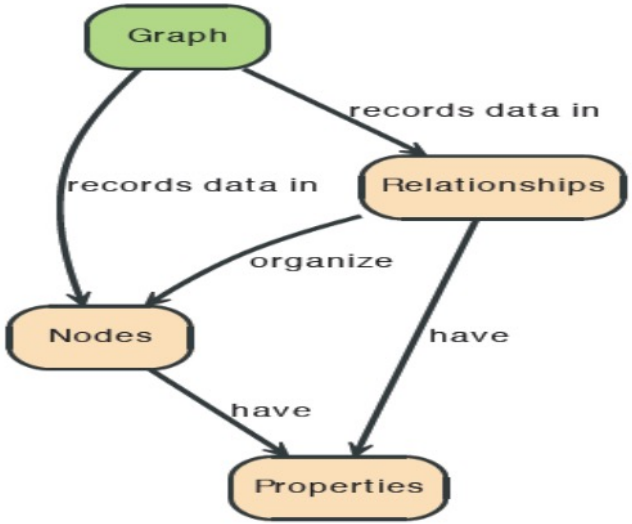
- Extremely fast for some queries and data models.
- Implement a language that vastly simplifies writing queries.

What are graphs good for?

- Recommendations
- Business intelligence
- Social computing
- Geospatial
- Systems management
- Web of things
- Genealogy
- Time series data
- Product catalogue
- Web analytics
- Scientific computing (especially bioinformatics)
- Indexing your *slow* RDBMS
- And much more!

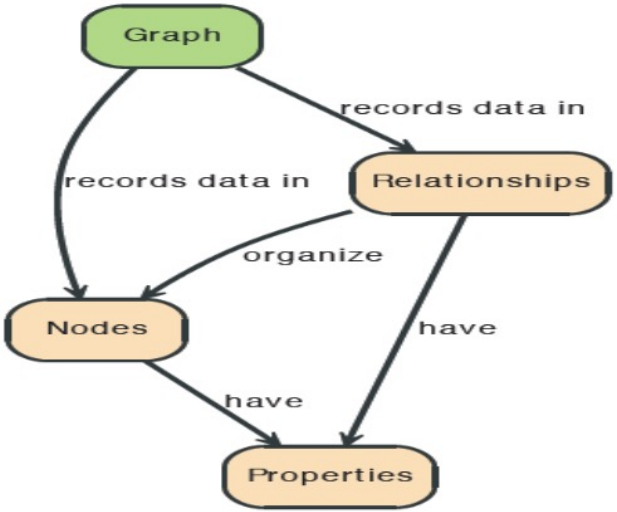
Graphs

- "A Graph —records data in→ Nodes —which have→ Properties"



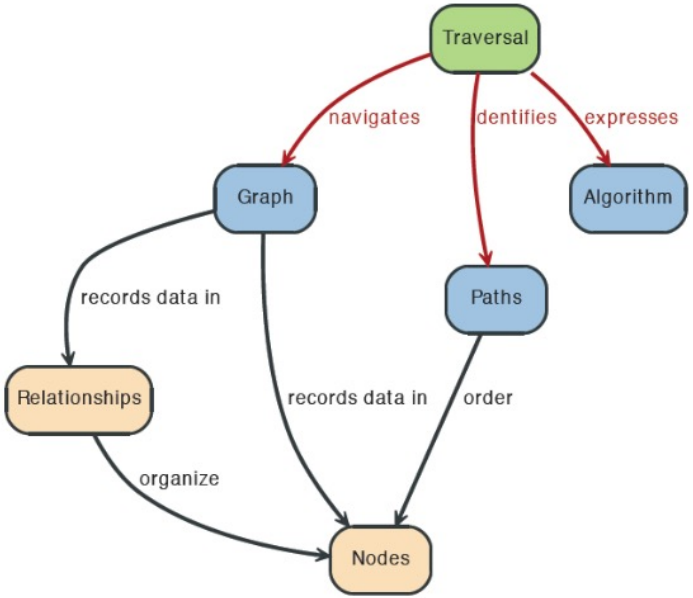
Graphs

- "Nodes —are organized by→ Relationships — which also have→ Properties"



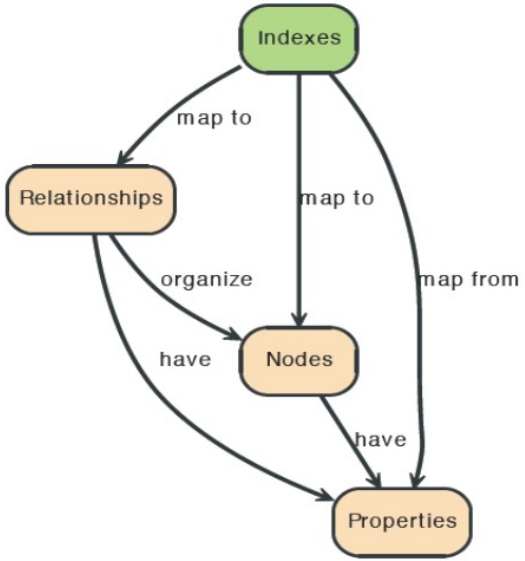
Query a graph with Traversal

- "A Traversal —navigates→ a Graph; it — identifies→ Paths —which order→ Nodes"

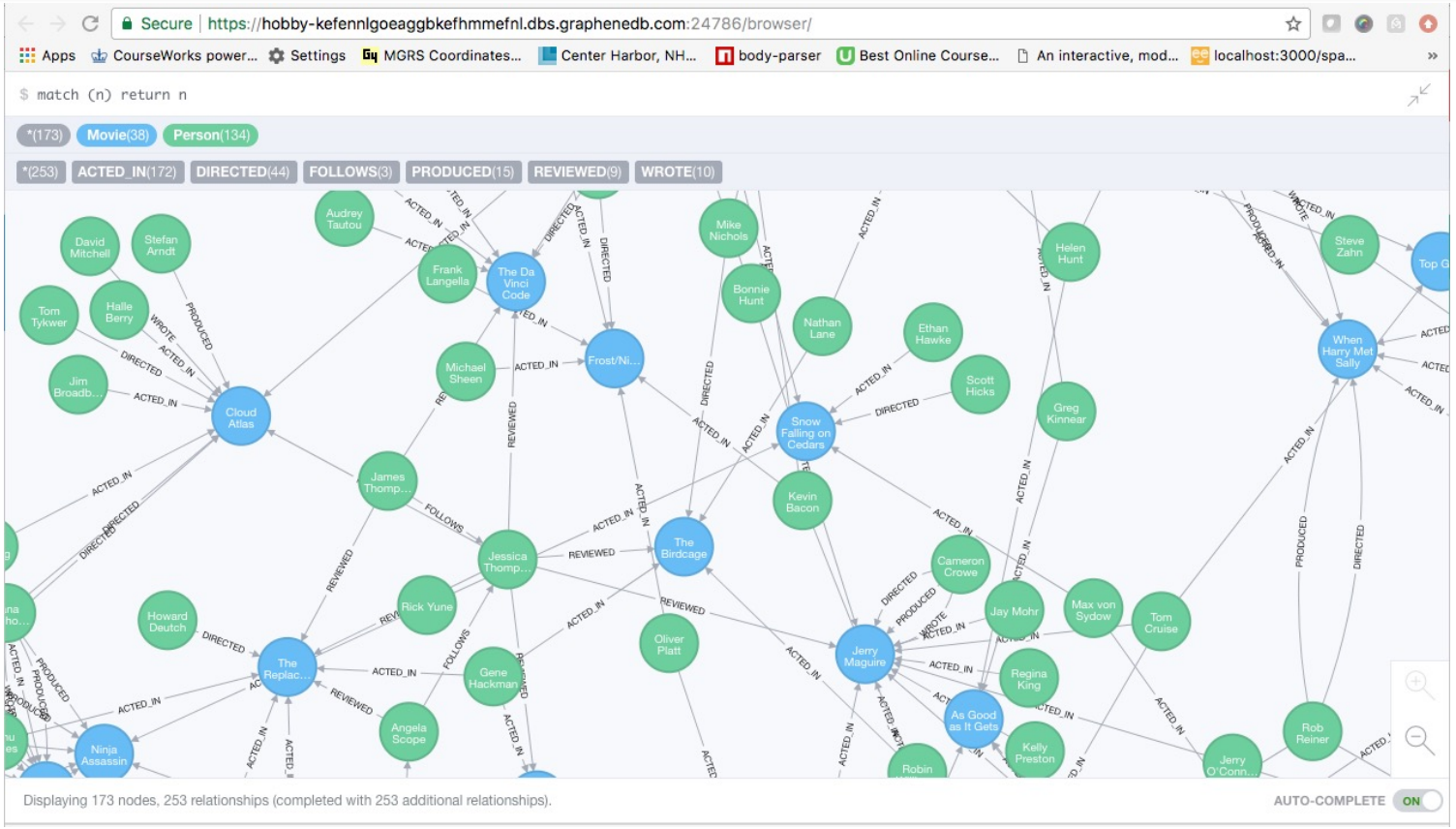


Indexes

- "An Index —maps from→ Properties —to either→ Nodes or Relationships"

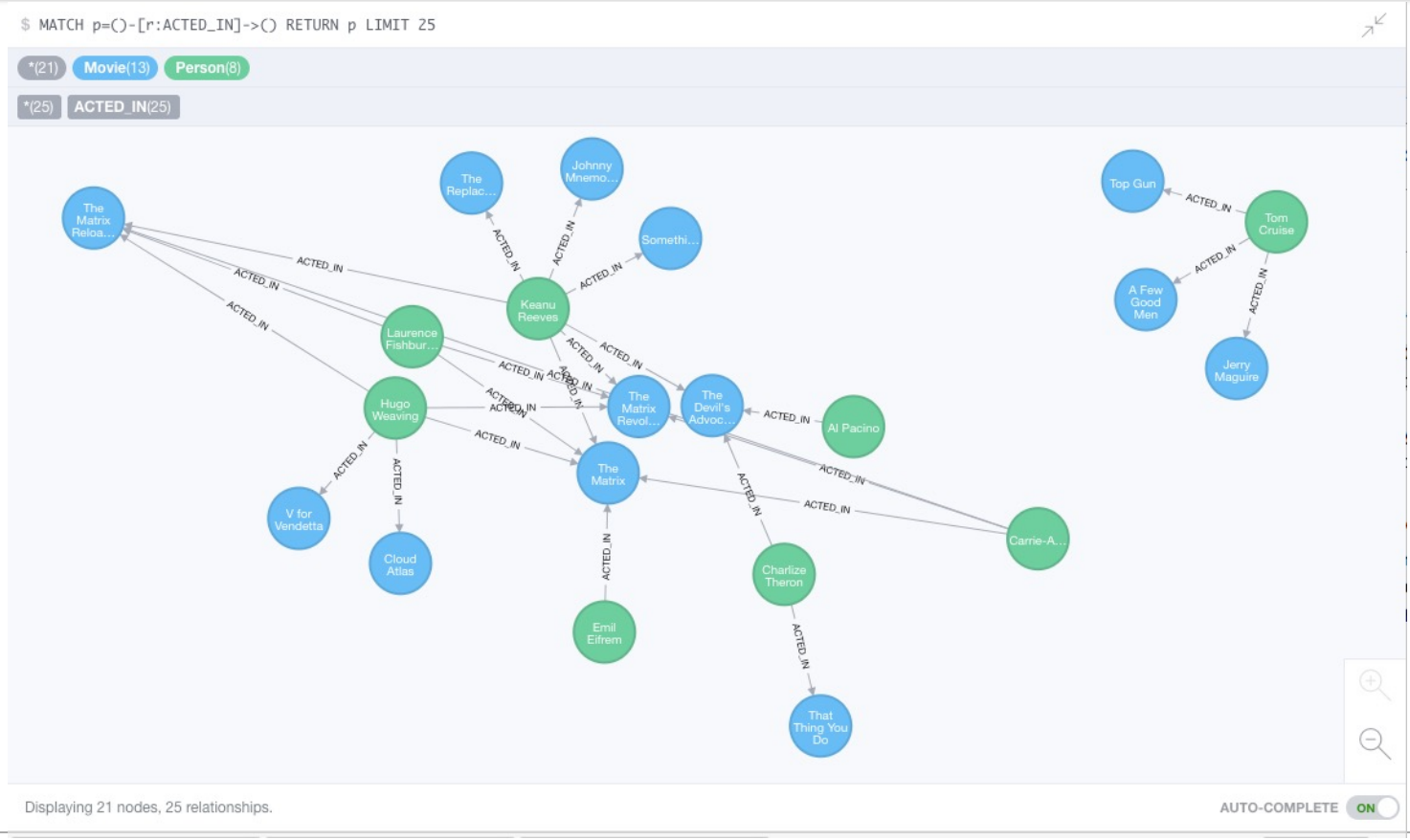


A Graph Database (Sample)



Neo4J Graph Query

Who acted in which movies?



Big Deal. That is just a JOIN.

- Yup. But that is simple.
- Try writing the queries below in SQL.

The Movie Graph

Recommend

Let's recommend new co-actors for Tom Hanks. A basic recommendation approach is to find connections past an immediate neighborhood which are themselves well connected.

For Tom Hanks, that means:

1. Find actors that Tom Hanks hasn't yet worked with, but his co-actors have.
2. Find someone who can introduce Tom to his potential co-actor.

Extend Tom Hanks co-actors, to find co-co-actors who haven't work with Tom Hanks...

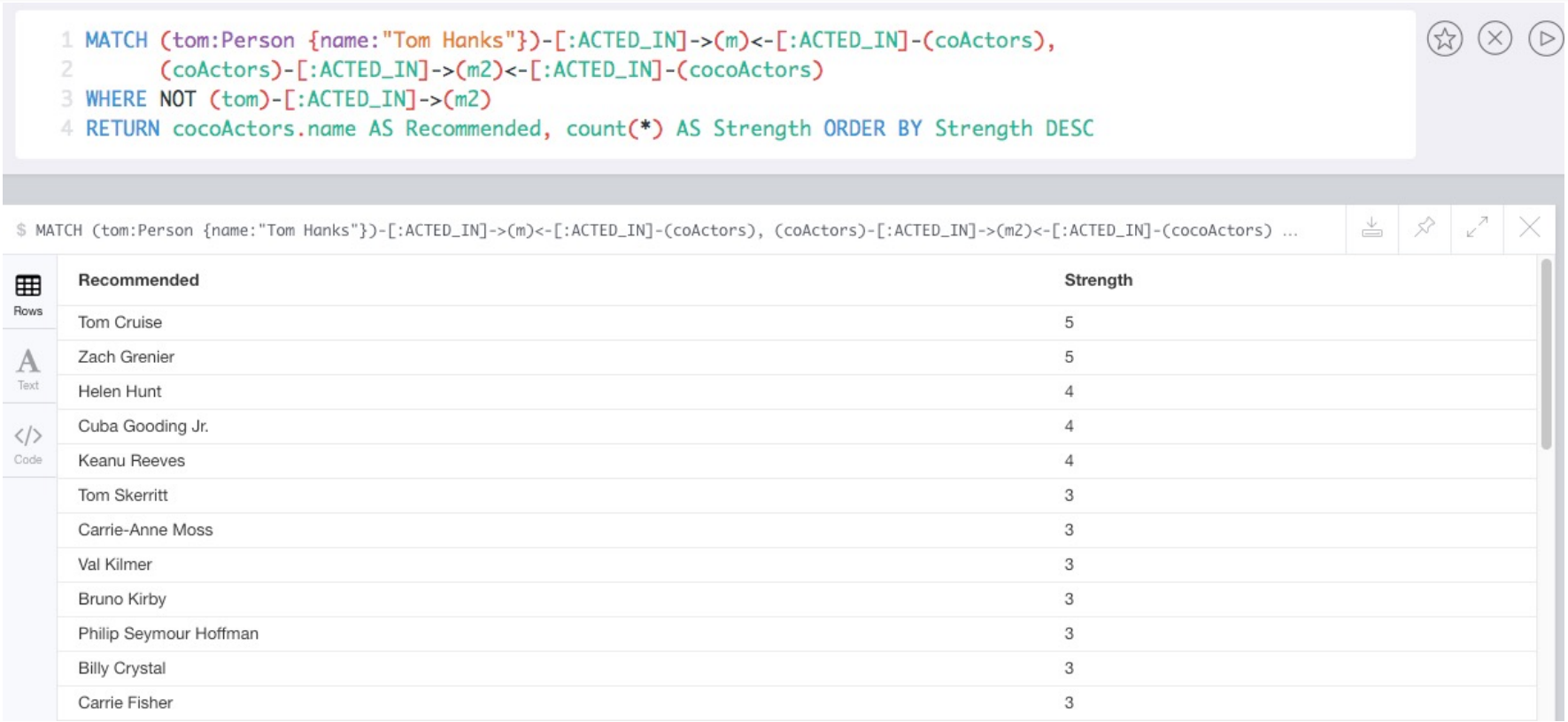
```
MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-(coActors),
      (coActors)-[:ACTED_IN]->(m2)<-[:ACTED_IN]-(cocoActors)
WHERE NOT (tom)-[:ACTED_IN]->(m2)
RETURN cocoActors.name AS Recommended, count(*) AS Strength ORDER BY Strength DESC
```

Find someone to introduce Tom Hanks to Tom Cruise

```
MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-(coActors),
      (coActors)-[:ACTED_IN]->(m2)<-[:ACTED_IN]-(cruise:Person {name:"Tom Cruise"})
RETURN tom, m, coActors, m2, cruise
```

Recommend

```
1 MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-(coActors),
2       (coActors)-[:ACTED_IN]->(m2)<-[:ACTED_IN]-(cocoActors)
3 WHERE NOT (tom)-[:ACTED_IN]->(m2)
4 RETURN cocoActors.name AS Recommended, count(*) AS Strength ORDER BY Strength DESC
```

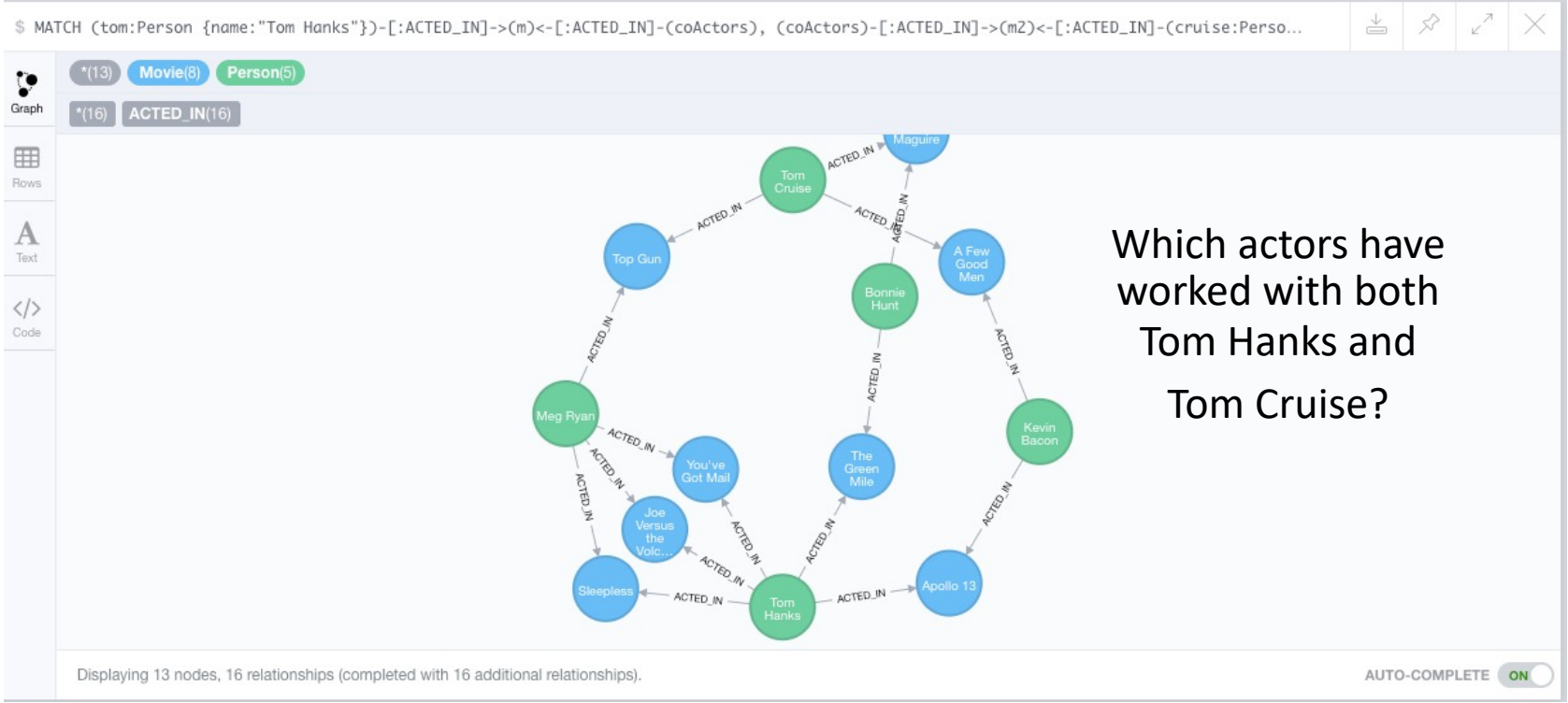


$ MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-(coActors), (coActors)-[:ACTED_IN]->(m2)<-[:ACTED_IN]-(cocoActors) ...

Rows | Text | Code

Recommended	Strength
Tom Cruise	5
Zach Grenier	5
Helen Hunt	4
Cuba Gooding Jr.	4
Keanu Reeves	4
Tom Skerritt	3
Carrie-Anne Moss	3
Val Kilmer	3
Bruno Kirby	3
Philip Seymour Hoffman	3
Billy Crystal	3
Carrie Fisher	3

```
MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-(coActors),
      (coActors)-[:ACTED_IN]->(m2)<-[:ACTED_IN]-(cruise:Person {name:"Tom Cruise"})
RETURN tom, m, coActors, m2, cruise
```

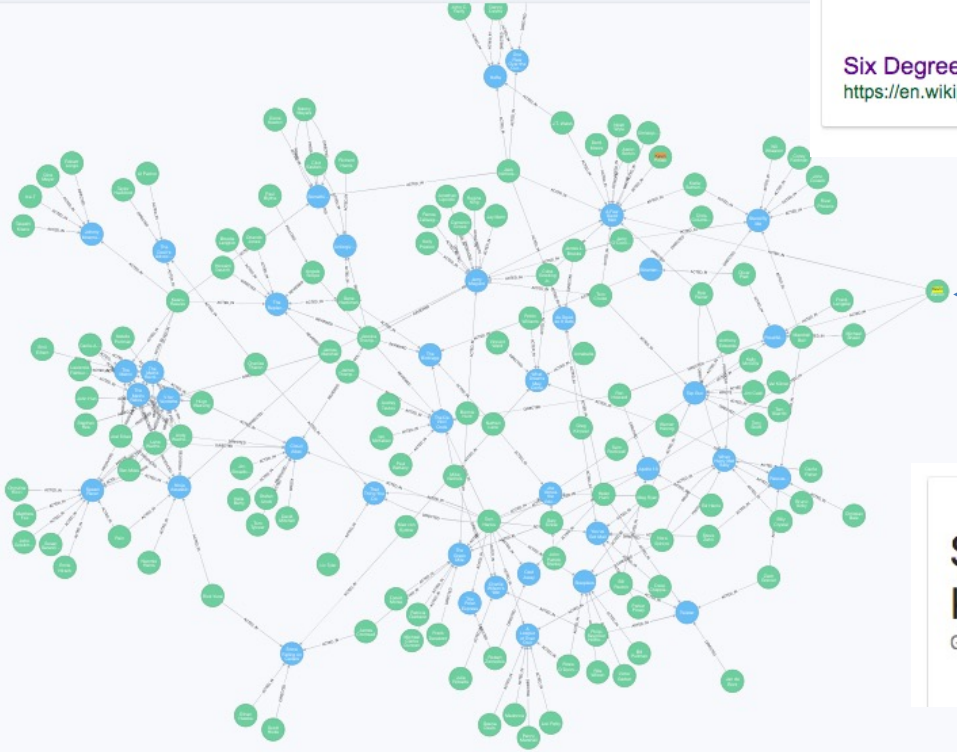


Which actors have worked with both Tom Hanks and Tom Cruise?

```
$ MATCH (s:Person { name: 'Kevin Bacon' })-[*0..6]-(m) return s,m
```

*(171) Movie(38) Person(133)

*(253) ACTED_IN(172) DIRECTED(44) FOLLOWS(3) PRODUCED(15) REVIEWED(9) WROTE(10)

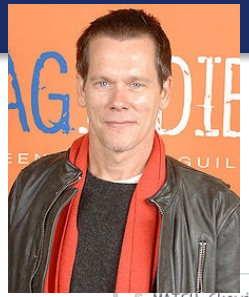


Six **Degrees of Kevin Bacon** is a parlour game based on the "six **degrees** of separation" concept, which posits that any two people on Earth are six or fewer acquaintance links apart. Movie buffs challenge each other to find the shortest path between an arbitrary actor and prolific actor **Kevin Bacon**.

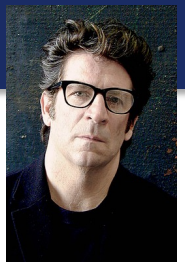
Six Degrees of Kevin Bacon - Wikipedia
https://en.wikipedia.org/wiki/Six_Degrees_of_Kevin_Bacon

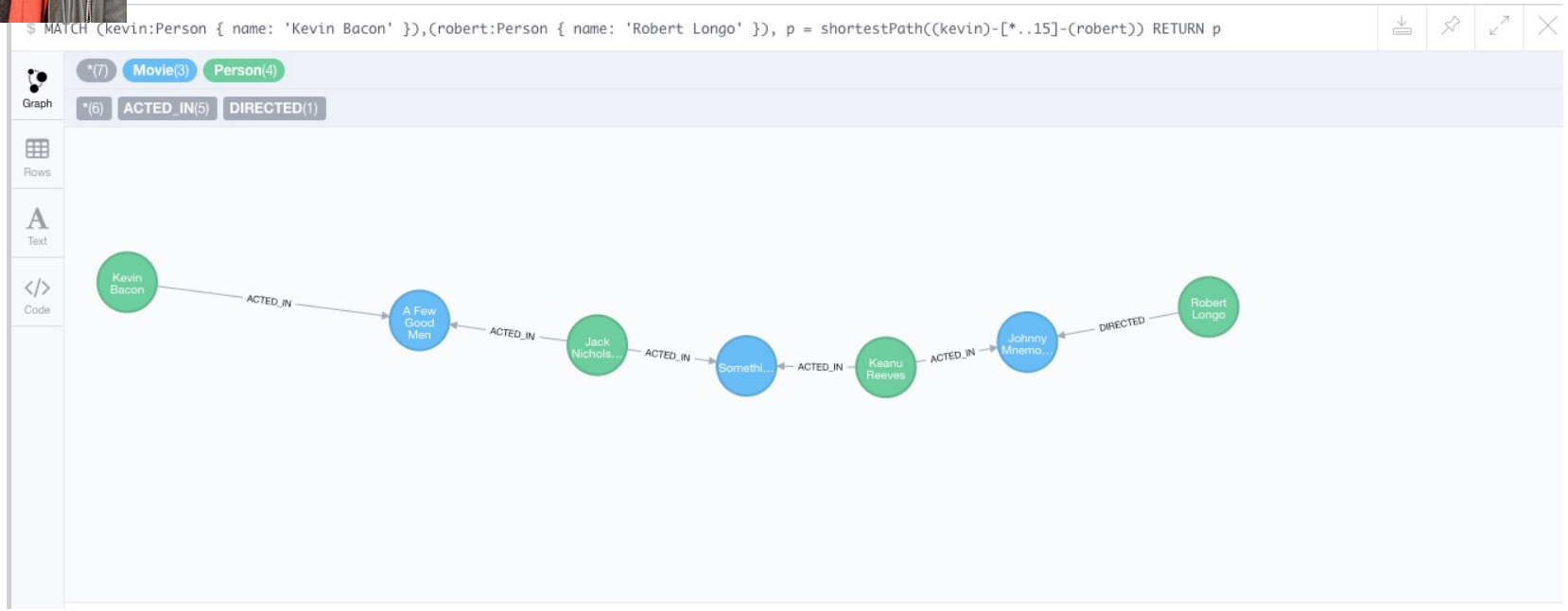
About this result Feedback

Six Degrees of Kevin Bacon
Game



How do you get from Kevin Bacon to Robert Longo?





Take a Test Drive
Neo4j

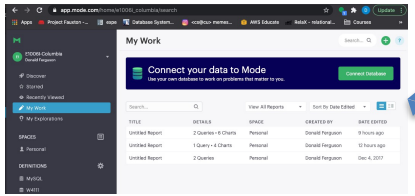
HW3 (Project)

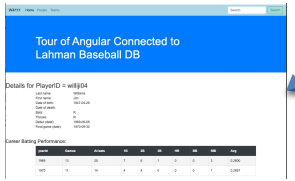
To Be Refined and Simplified!

Pretend to Play Fantasy Baseball

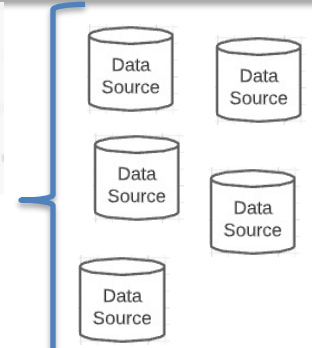
- (My) Project for the two tracks will be a **simple** fantasy baseball solution.
  - "**Fantasy baseball** is a game in which people manage rosters of league baseball players, either online or in a physical location, using fictional fantasy baseball team names. The participants compete against one another using those players' real-life statistics to score points."
- My solution will have two subsystems:
  - *Analysis system*:
    - Read only data that enables people to select players based on performance, and to estimate the performance of their fantasy team.
    - Event based update system that changes analysis data based on new data.
  - *Operational system* that manages create, retrieve update, delete, etc. of their fantasy teams and leagues.
- INSERT, UPDATE, DELETE primarily apply to the operational system.

Target Application/Project(s) – Reminder (Lecture 1)





Data Insight Service



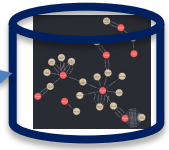
Web Application

Social Network Service

Core Data Service

Comment Discussion Service

DB Access Service



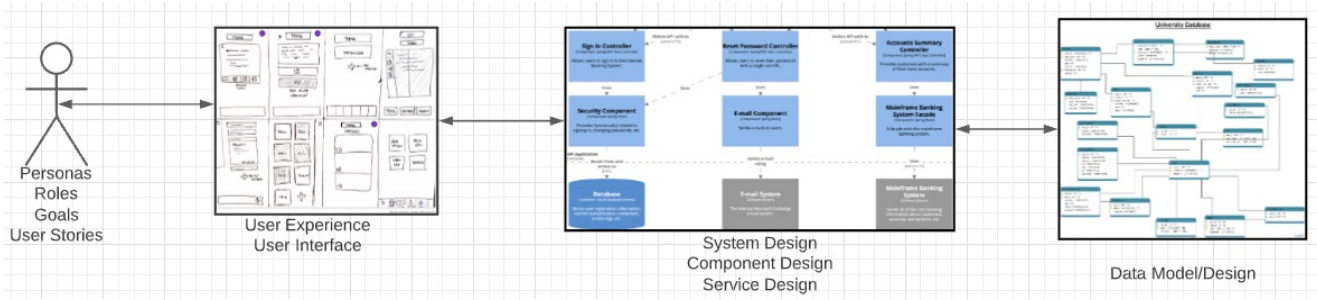
Target Application/Project – Reminder (Lecture 1)

- That diagram was pretty confusing.
- Basically, what it comes down to is that there will be major subsystems:
  1. Interactive web application for viewing, navigating and updating data.
     The updates have to preserve *semantic constraints* and correctness.
  2. A decision support warehouse/lake that allows us to explore data and get insights.
- Programming and non-programming tracks will get experience with both, but
  - Non-programming track focuses on data engineering needed to produce (2).
  - Programming track will focus on (1).
- We will use is *fantasy baseball* because:
  - It has aspects and tasks interesting to both tracks.
  - We have an existing data set that we have been using.
  - There are interesting additional sources of data and use cases.

Operational System (Web Application)

Problem Statement – Modified (Lecture 1 Reminder)

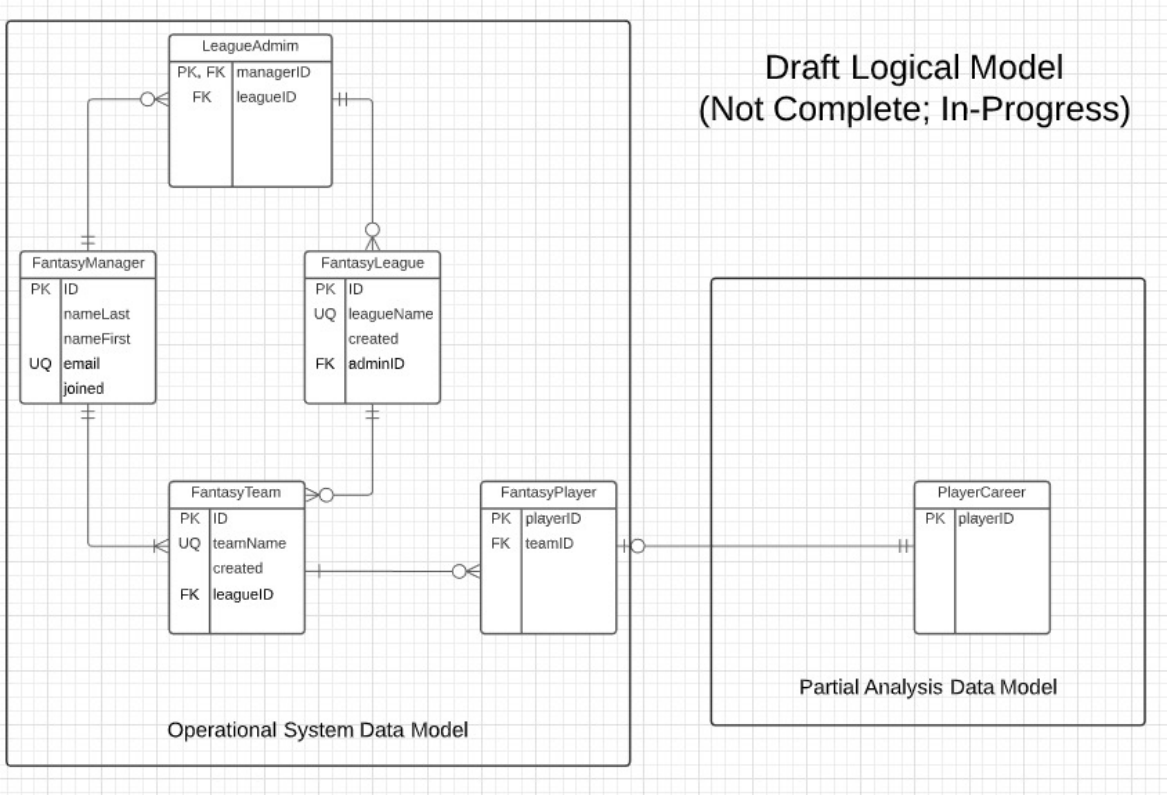
- We must build a system that supports … … operations in a … …
- We will design, develop, test and deploy the system iteratively and continuously.
- There are four core domains.



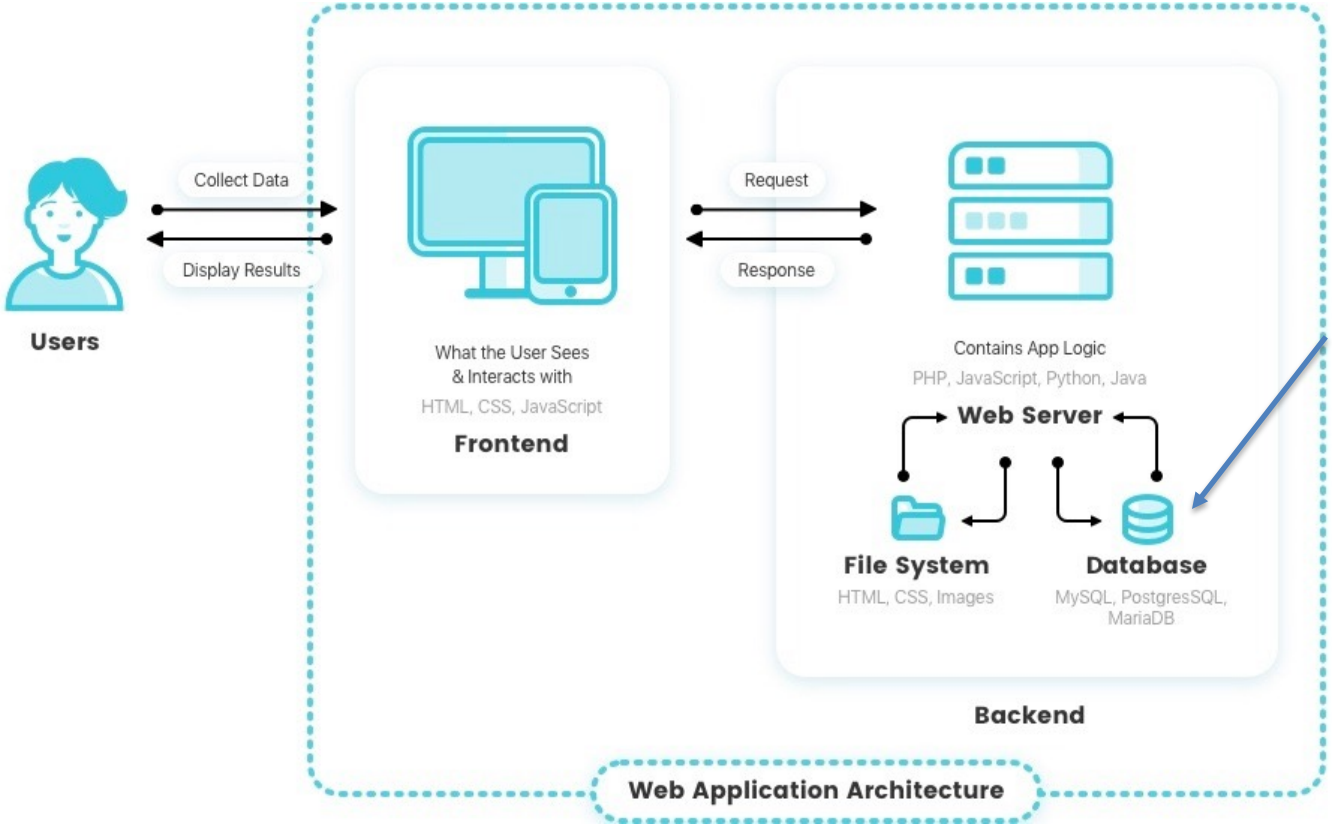
- In this course,
- We focus on the data dimension.
- We will get some insight into the other dimensions.

- The processes are iterative, with continuous extension and details.
- We will start implementing various *user stories.* Implementation requires:
  - Web UI
  - Paths
  - Data model and operations.

In-Progress Operational System Data Model



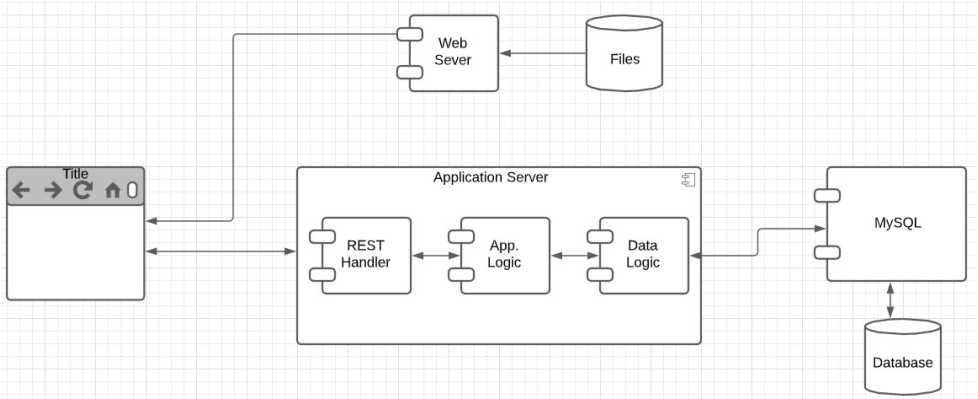
Web Application – Operational System



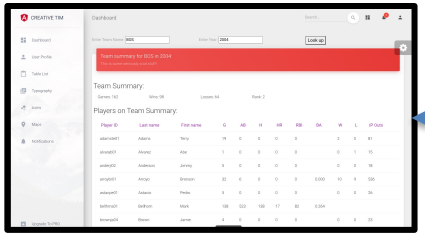
Lahman's Baseball DB
Fantasy Baseball DB

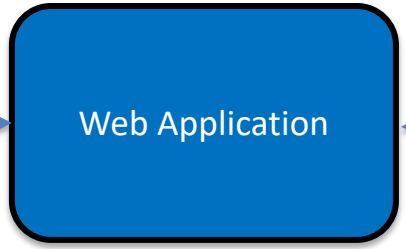
User Story

- “In software development and product management, a user story is an informal, natural language description of one or more features of a software system. User stories are often written from the perspective of an end user or user of a system. ” (https://en.wikipedia.org/wiki/User_story)
- Example user stories that I need to implement for the operational system:
  - ”As a fantasy team manager, I want to search for players based on career stats.”
  - “As a fantasy team manager, I want to add a player to my fantasy team.
  - etc.
- I need to implement:
  - UI
  - Application logic
  - Database tables.



Simple Example





Two HTTP/REST calls:

- GET /api/team_summary?team_id=BOS&year_id=2004
- GET /api/teams?teamID=BOS&yearID=2004

Two SQL Queriess:

- SELECT to Teams table.
- SELECT to team_summary, which is a view. We cover views later.

- "As a baseball fan, I want to enter a teamID and yearID and see:
  - Team wins and losses.
  - Summary of player performance for players on the team and year."
- Built:
  - Dashboard page.
  - REST handlers and application logic.
  - Database view.

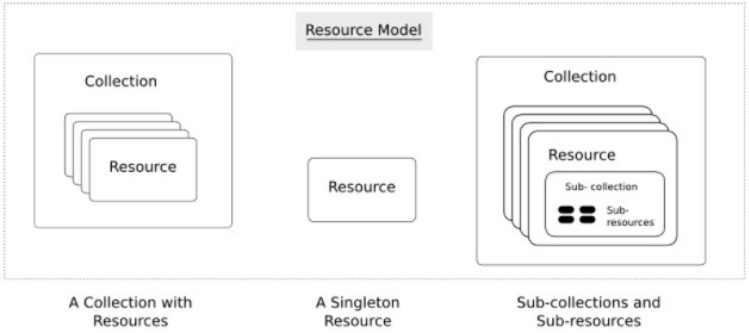
Tracks will build:

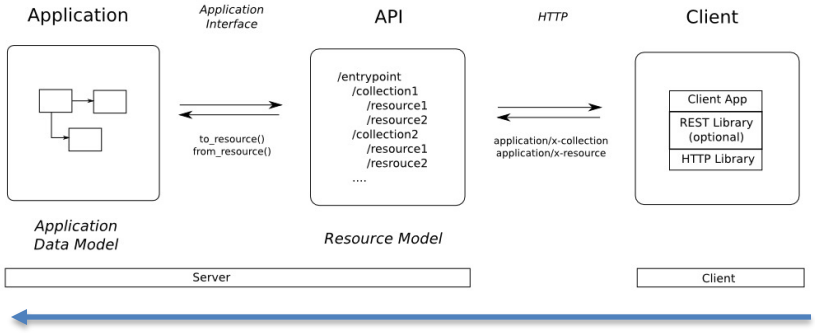
- Programming: Mix of CRUD and dashboard.
- Non-programming: Dashboards, Data transformation.

Will see more details as we move forward.

Simple Example – Fantasy Team Resource

https://restful-api-design.readthedocs.io/en/latest/resources.html





Interaction flow is right-to-left in this diagram.

- Resources: (https://restful-api-design.readthedocs.io/en/latest/resources.html)
  - The fundamental concept in any RESTful API is the resource. A resource is an object with a type, associated data, relationships to other resources, and a set of methods that operate on it.
  - ... ... information that describes available resources types, their behavior, and their relationships the resource model of an API. The resource model can be viewed as the RESTful mapping of the application data model.
- APIs:
  - ... ... APIs expose functionality of an application or service that exists independently of the API. (DFF comment – the data)
  - Understanding enough of the important details of the application for which an API is to be created, so that an informed decision can be made as to what functionality needs to be exposed ... ...
  - Modeling this functionality in an API that addresses all use cases that come up in the real world ... ...

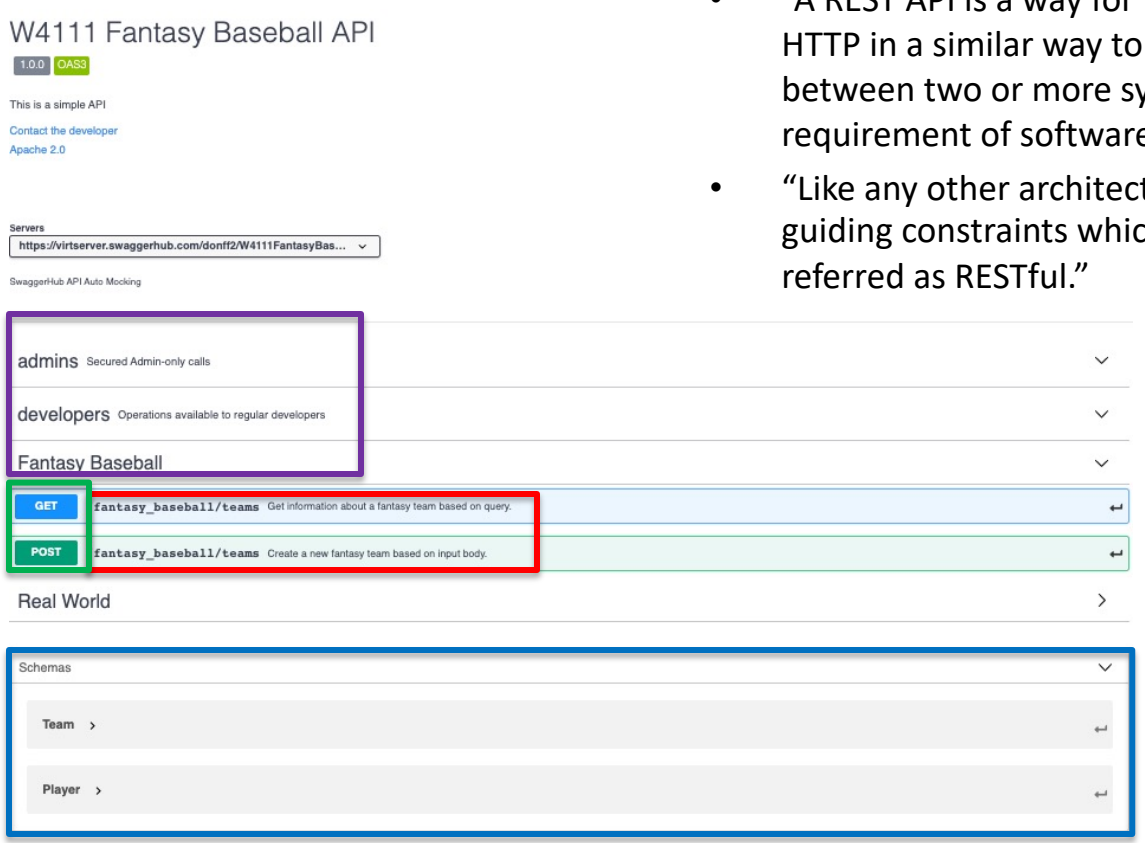
CRUD (https://en.wikipedia.org/wiki/Create,_read,_update_and_delete)

- Definitions:
  - "In computer programming, create, read, update, and delete[1] (CRUD) are the four basic functions of persistent storage."
  - "The acronym CRUD refers to all of the major functions that are implemented in relational database applications. Each letter in the acronym can map to a standard Structured Query Language (SQL) statement, Hypertext Transfer Protocol (HTTP) method (this is typically used to build RESTful APIs[5]) or Data Distribution Service (DDS) operation."

CRUD	SQL	HTTP	DDS
create	INSERT	PUT	write
read	SELECT	GET	read
update	UPDATE	PUT	write
delete	DELETE	DELETE	dispose

- Do not worry about Data Distribution Service.
- For our purposes HTTP – REST
- Entity Set:
  - Table in SQL
  - Collection Resource in REST.
- Entity:
  - Row in SQL
  - Resource in REST

REST API Definition



- "A REST API is a way for two computer systems to communicate over HTTP in a similar way to web browsers and servers. Sharing data between two or more systems has always been a fundamental requirement of software development.)
- "Like any other architectural style, REST also does have it's own 6 guiding constraints which must be satisfied if an interface needs to be referred as RESTful."

Open API Definition

- API Tags/Groupings
- A resource has
  - Paths
  - Methods
- Schema (Data Formats)
  - Sent on POST and PUT
  - Returned on GET

This material is just FYI and to help with understanding concepts, mapping to DB, ...

Demo and Next Steps

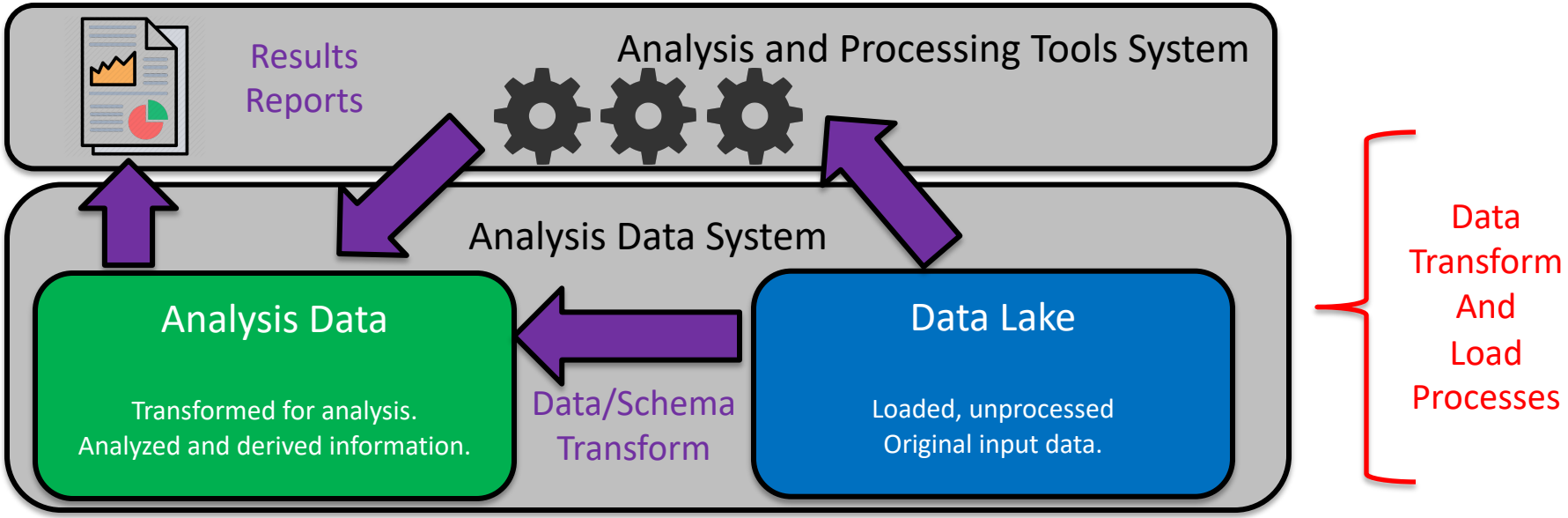
- API: https://app.swaggerhub.com/apis/donff2/W4111FantasyBaseball/1.0.0#/
- Starter API project:
  - App (Postman) http://127.0.0.1:5000/api/fantasy_team/BOS20011
  - Project: ~/Dropbox/Columbia/W4111_S21_New/W4111S21/Projects/FantasyBaseball

- Next Steps:
  - Will commit the template project, sample code and schedule orientation in recitation.
  - Will try to provide some UI, but this is not a UI class.

Analysis System

Analysis System – Fantasy Baseball Concept

- Focus is on the Analysis Data System (Primary Focus):
  - Data Lake is source data, imported and added to common database/model. (e.g. Lahman Baseball DB)
  - Analysis data is transformed data suitable for analysis, and analysis results. (e.g. Transformed Lahman's Data)
- Various analysis and processing tools use the data for insight, visualization, etc. (e.g. Jupyter, Pandas)



Data Lake to Analysis Data

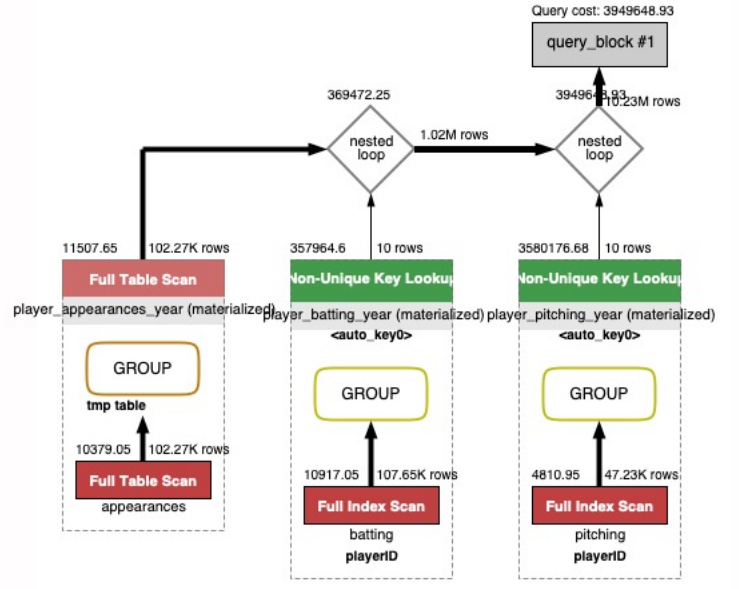
- This is a database class. So, the raw data to analysis data is a focus.
- Consider the following requirement.
  - I want to be able to predict a player's performance relative to salary based on history.
  - The first step is joining information from several tables, e.g.
    - Appearances.
    - Batting.
    - Pitching.
    - ~~Fielding~~. We are not trying to be super accurate about baseball. This is a DB class. We will ignore fielding.
    - Salary.
  - We then need to produce a single summary/prediction row. But, this is not as simple as just summing and averaging.
    - A year's numbers may be low because the player was injured and did not play many games.
    - The average is not meaningful. Over their career, player's get better and then worse. We want to use averages for historical players but trends for active players.
    - Salary data is missing. Average salary changed over time and we need to normalize.
    - **Absolute numbers do not matter. What matters is relative performance to other players**.

- **We will not do these now.**
- **I will do during other lectures and as I do my project.**

- Let's just do a very simple example. Career summary of annual performance relative to averages.

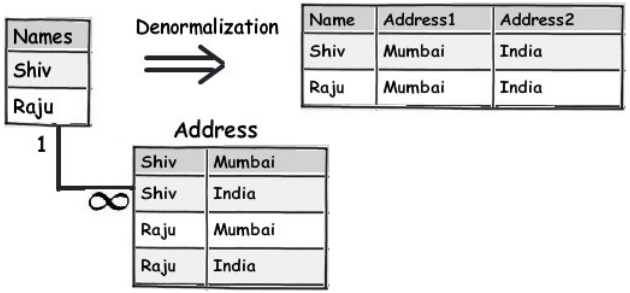
Task 1 – Player, Year Summary

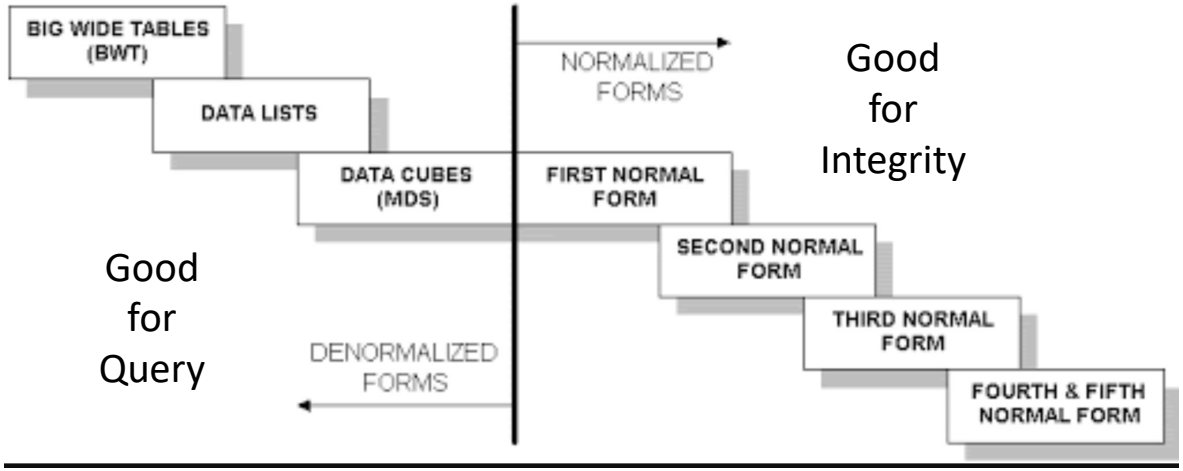
```
with player_basic
  as (
    select playerID, nameLast, nameFirst, bats, throws from people),
   player_appearances_year as
   (
     select
       playerid, yearid, sum(g_all) as app_g_all, sum(gs) as app_g_s, sum(g_defense) as app_g_defense,
       sum(g_p) as app_g_p, sum(g_c) as app_g_c, sum(g_1b) as app_g_1b, sum(g_2b) as app_g_2b,
         sum(g_3b) as app_g_3b, sum(g_ss), sum(g_lf) as app_g_lf, sum(g_cf) as app_g_cf,
           sum(g_rf) as app_g_rf
     from appearances group by playerid, yearid
   ),
   player_batting_year
  as
   (
     select playerid, yearid, sum(g) as bat_g, sum(g_batting) as g_batting,
          sum(ab) as bat_ab, sum(r) as bat_r, sum(h) as bat_h,
          sum(h-`2b`-`3b`-hr) as bat_1, sum(`2b`) as bat_2, sum(`3b`) as bat_3,
          sum(hr) as bat_hr, sum(rbi) as bat_rbi, sum(sh) as bat_sh, sum(sf) as bat_sf,
         sum(`2b`) as b2, sum(`3b`) as b3, sum(hr) as hr, sum(bb) as bb
    from batting group by playerid, yearid
  ),
  player_pitching_year
  as
    (
    select playerid, yearid, sum(w) as p_w, sum(l) as p_l, sum(g) as p_g, sum(IPOuts) as p_IPOuts,
      sum(SV) as p_SV, sum(h) as p_h, sum(hr) as p_hr, sum(bb) as p_bb, sum(er) as p_er
      from pitching group by playerid, yearid
      )
select * from
  (select * from player_appearances_year left join player_batting_year using(playerid, yearid)) as x
  left join
  player_pitching_year using(playerid, yearid)
```



- The core of the data lake are the raw tables.
- We clearly do not want to run this query every time we decide to examine and choose players.
- Task 1 adds a processed, “wide flat” table to the data lake for subsequent processing.

Wide Flat Tables





- Improve query performance by precomputing and saving:
  - JOINs
  - Aggregation
  - Derived/computed columns
- One of the primary strength of the relational model is maintaining “integrity” when applications create, update and delete data. This relies on:
  - The core capabilities of the relational model, e.g. constraints.
  - A well-design database (We will cover a formal definition – “normalization” in more detail later.
- Data models that are well designed for integrity are very bad for read only analysis queries.
  We will build and analyze wide flat tables as part of the analysis tasks in HW3, HW4 as projects.

External Information

- Analysis projects often include data from multiple sources and domains.
- Consider an example:
  - Building the best “fantasy baseball team” is an optimization problem.
  - Find the “best” players at the “lowest price (salary).”
- We have a few problems:
  - We must normalize old values to modern (constant) dollars.
  - There has been a general, upward trend in salaries. What matters is a player’s salary relative to the average salary for a year.
  - We have “missing” salaries:
    - No information before 1985.
    - Missing some entries in the range 1985-present.
- We will just do a hypothetical normalization right now.

Normalize the Value of the Dollar

- "A constant dollar is an adjusted value of currency used to compare dollar values from one period to another."
  (Investopedia)
- "To accurately compare income over time, users should adjust the summary measures (medians, means, etc.) for changes in cost of living (prices)."
  (US Census).
- This is not an MBA or economics class.
  We are just going to do something hypothetical and simple.
- "To use the CPI-U-RS to inflation adjust an income estimate from 1995 dollars to 2019 dollars, multiply the 1995 estimate by the CPI-U-RS from 2019 (376.5) divided by the CPI-U-RS from 1995 (225.3)"
  (US Census)

	CPI_US_I	Value
47	1993	215.5
48	1994	220
49	1995	225.3
50	1996	231.3
51	1997	236.3
52	1998	239.5
53	1999	244.6
54	2000	252.9
55	2001	260.1
56	2002	264.2
57	2003	270.2
58	2004	277.5
59	2005	286.9
60	2006	296.2
61	2007	304.6
62	2008	316.3
63	2009	315.2
64	2010	320.4
65	2011	330.5
66	2012	337.5
67	2013	342.5
68	2014	348.3
69	2015	348.9
70	2016	353.4
71	2017	361
72	2018	369.8
73	2019	376.5

- A 1995 salary of $100,000 is $ (376.5 / 225.3)*100000 = $167110.
- We will compute the inflation adjusted, average annual salary.

Hypothetical Normalization

```
with annual_average_salary as
  (
    select yearid, round(avg(salary),0) as avg_salary from
      lahmansbaseballdb.salaries group by yearid
  ),
normalized_averages as
      (
        select yearid, avg_salary, cpi.value  as salary_factor,
            round((avg_salary / cpi.value), 0) as normalized_avg_salary
        from annual_average_salary
            join cpi on yearid=cpi.CPI_US_I
      ),
averages_and_yoy as
  (
    select a.yearid, a.avg_salary, a.salary_factor, b.avg_salary as prior_avg_salary,
        b.salary_factor as prior_salary_factor
      from normalized_averages as a join normalized_averages as b
        on a.yearid=b.yearid+1
  )
select yearid, avg_salary salary_factor,
    round(avg_salary/prior_avg_salary,3) avg_salary_yoy,
     round(salary_factor/prior_salary_factor,3) as salary_factor_yoy
      from averages_and_yoy
order by yearid desc;
```

	yearid	salary_factor	avg_salary_yoy	salary_factor_yoy
1	2016	4396410	1.022	1.013
2	2015	4301276	1.081	1.002
3	2014	3980446	1.069	1.017
4	2013	3723344	1.077	1.015
5	2012	3458421	1.042	1.021
6	2011	3318838	1.012	1.032
7	2010	3278747	1	1.016
8	2009	3277647	1.045	0.997
9	2008	3136517	1.066	1.038
10	2007	2941436	1.038	1.028
11	2006	2834521	1.076	1.032
12	2005	2633831	1.057	1.034
13	2004	2491776	0.968	1.027

- I doubt very much that I got this correct.
- But you get the basic idea:
  - Process core data from the domain.
  - Bring in external information to adjust.
- Bring this together with the larger data, in this case to be able to talk about player performance relative to salary.

Hypothetical Player Choice

- Scenario:
  - I need a catcher for my team.
  - I want to find the “best 5 batting catchers by year.”
  - I measure batting performance by *slugging percentage.*
  - I want to normalize salary relative averages and CPI.
  - I do not want *absolute best.* I want best price performance, e.g. *(slugging percentage)/(normalized_salary)*

Switch to Notebook.

	yearid	playerid	nameLast	nameFirst	slg	relative_salary	slg_per_salary
1	2006	mccanbr01	McCann	Brian	0.5724	0.118	4.851
2	2001	pierzaj01	Pierzynski	A. J.	0.4409	0.092	4.793
3	2000	melusmi01	Meluskey	Mitch	0.4866	0.108	4.506
4	2000	estalbo02	Estalella	Bobby	0.4682	0.105	4.459
5	1993	piazzmi01	Piazza	Mike	0.5612	0.129	4.351

Optimal Price/Performance Catcher

```
with annual_batting_performance_salary as
     (
       select playerid,
            yearid,
            app_g_c,
            (bat_h / if(bat_ab = 0, NULL, bat_ab))                    as bat_avg,
            ((bat_h + bb) / (bat_ab + bb))                          as bat_obp,
            ((bat_1 + bat_2 * 2 + bat_3 * 3 + bat_hr * 4) / if(bat_ab = 0, NULL, bat_ab)) as slg,
            salary
       from player_performance_year
             join
           (select playerid, yearid, avg(salary) as salary
            from lahmansbaseballdb.salaries
            group by playerid, yearid) as a
           using
             (playerid, yearid)
       where bat_ab >= 100
     ),
normalized_salary_batting_performance as
  (
    select yearid, playerid,  bat_obp, slg, app_g_c,
        round(salary/normalized_salary_averages.salary_factor,3) as relative_salary from
         annual_batting_performance_salary join
          normalized_salary_averages using(yearid)
  )
select
  yearid, playerid, nameLast, nameFirst, slg, relative_salary, round(slg/relative_salary,3) as slg_per_salary
from normalized_salary_batting_performance join lahmansbaseballdb.people using(playerid)
  where app_g_c > 100
  order by slg_per_salary desc
limit 5;
```

- If I got that correct, it was a miracle.
- The purpose wasn't to win at Fantasy Baseball.
- Just giving a feel for the types of things that
  - Will happen in HW3, HW4 (Project) in
  - Analysis Subsystem
- The non-programming track will use tools:
  - Jupyter
  - SciKit
  - … …
  - To analyze.
- This is a database track. So, the programming track will also get some experience.
- Process for both tracks:
  - You can pick your own data and domain
  - Or follow along with me.
- There will be a set of criteria that you have to meet for each of HW4 and HW4, measure in:
  - Data model complexity.
  - Complexity of operations on the data.

HW3, HW4 – The Project

Previous slides introduced the concepts.
Carrie and I plan to meet, simplify, refine, ...
based on experiences from last semester
and this semester.

Stay Tuned.