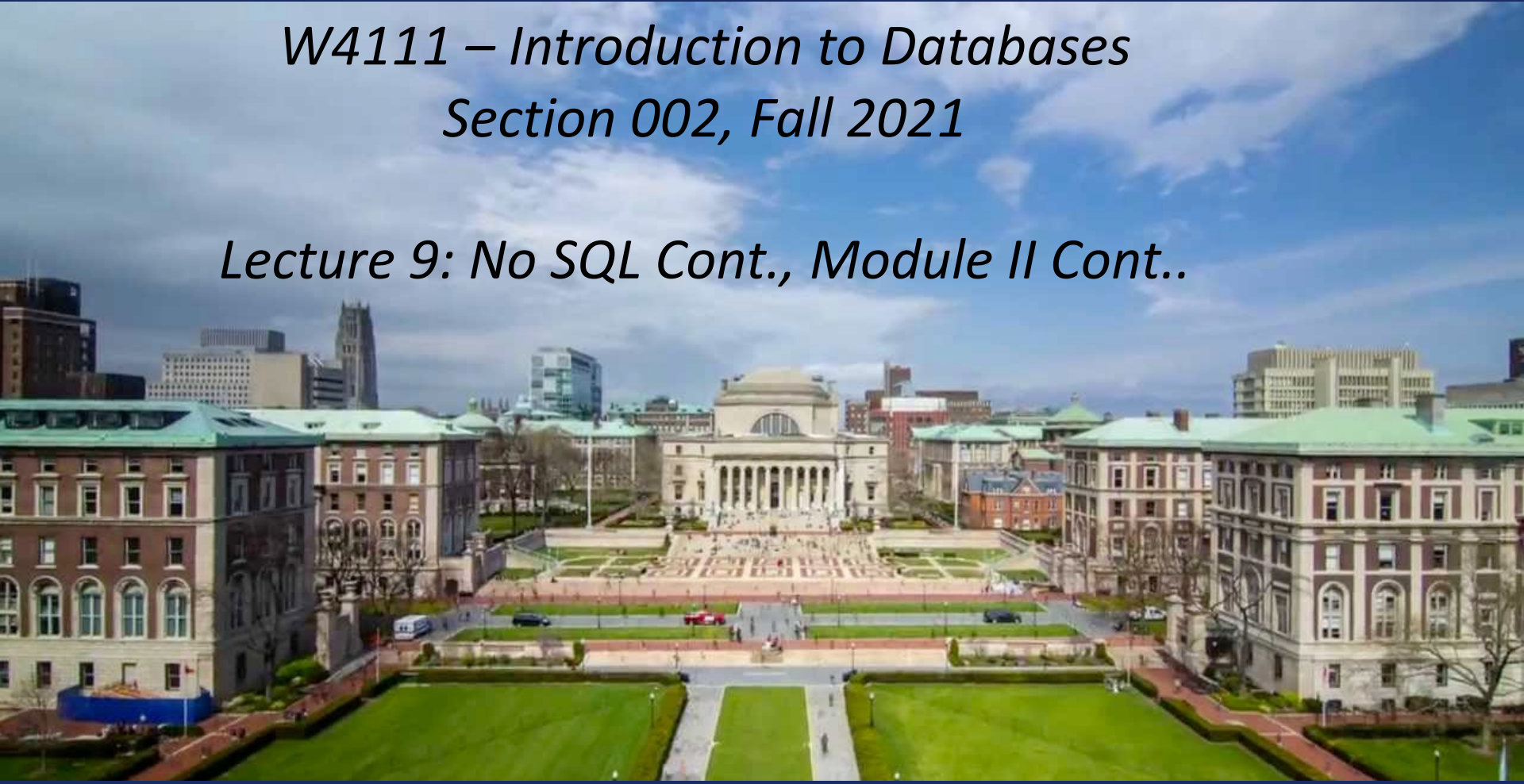
W4111 – Introduction to Databases
Section 002, Fall 2021

Lecture 9: No SQL Cont., Module II Cont..



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

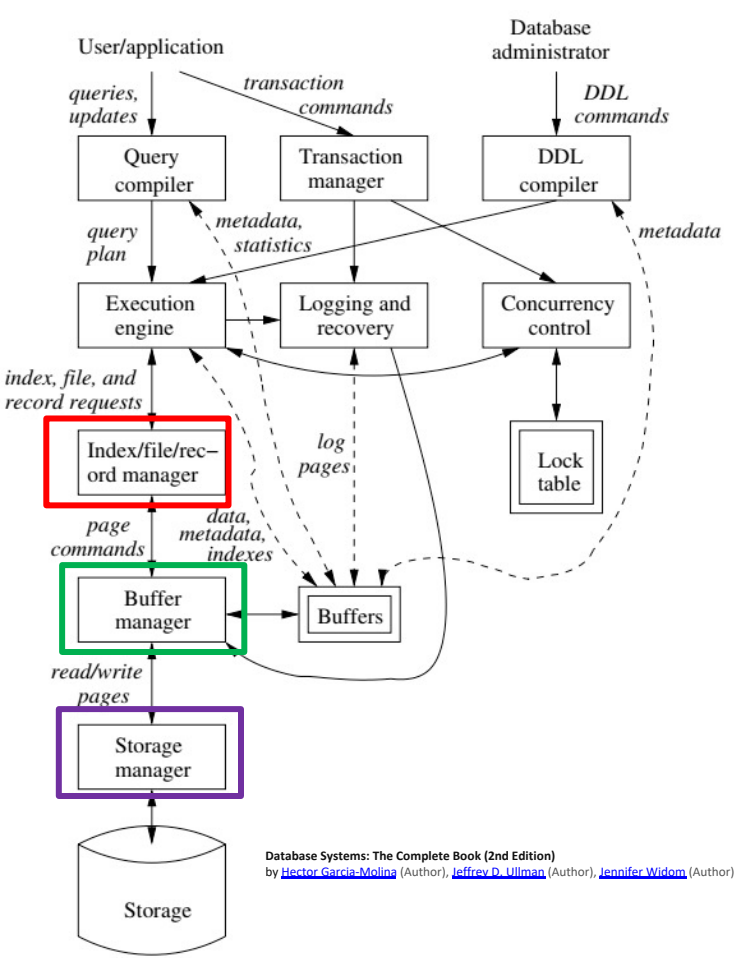
Contents

Module II – Continuation

Module II – DBMS Architecture and Implementation Overview and Reminder

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)

Data Storage Structures:
Very Simple Overview
(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

  .

On My Mac

mysql — -bash — 140×26

```
(base) MacBook-Pro-4:mysql donaldferguson$ pwd
/usr/local/mysql
(base) MacBook-Pro-4:mysql donaldferguson$ sudo ls data
#innodb_temp            aaaaCUClasses           binlog.000336           cu_info                 mysql
CSVCatalog              aaaaF21                 binlog.000381           cu_model                mysql.ibd
F21Midterm              aaaaW4111Example        binlog.000382           db_book                 mysqld.local.err
Foods                   aaaaaF21E6156           binlog.000383           ib_buffer_pool          mysqld.local.pid
GOT                     aaaaaF21UserAddress     binlog.000384           ib_logfile0             performance_schema
GOTProcessed            aaaaaNYG                binlog.000385           ib_logfile1             private_key.pem
IMDBFixed               auto.cnf                binlog.index            ibdata1                 public_key.pem
IMDBProcessed           binlog.000311           ca-key.pem              ibtmp1                  server-cert.pem
IMDBRaw                 binlog.000312           ca.pem                  lahmansbaseballdb       server-key.pem
Library                 binlog.000313           classicmodels           lahmansbaseballdb2019   undo_001
NYGBackup               binlog.000314           client-cert.pem         lahmansdb_to_clean      undo_002
NewYorkGuardNew         binlog.000334           client-key.pem          macbook-pro-4.lan.err   univ_example
(base) MacBook-Pro-4:mysql donaldferguson$ sudo ls data/lahmansbaseballdb
allstarfull.ibd                 divisions.ibd                   managers.ibd                    salaries.ibd
appearances.ibd                 fielding.ibd                    managershalf.ibd                schools.ibd
awardsmanagers.ibd              fieldingof.ibd                  parks.ibd                       seriespost.ibd
awardsplayers.ibd               fieldingofsplit.ibd             people.ibd                      teams.ibd
awardssharemanagers.ibd         fieldingpost.ibd                people_constraints.ibd          teamsfranchises.ibd
awardsshareplayers.ibd          full_names_and_initials.ibd     people_copy.ibd                 teamshalf.ibd
batting.ibd                     halloffame.ibd                  people_williams.ibd
battingpost.ibd                 homegames.ibd                   pitching.ibd
collegeplaying.ibd              leagues.ibd                     pitchingpost.ibd
(base) MacBook-Pro-4:mysql donaldferguson$
```

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

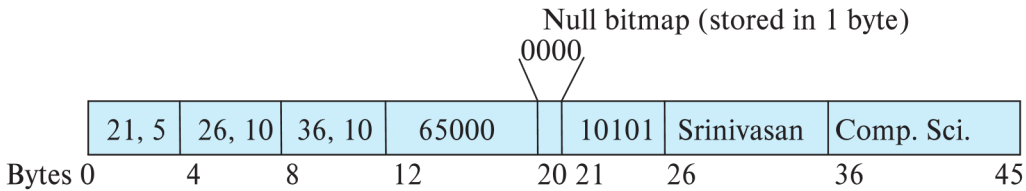
Fixed-Length Records

- 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***

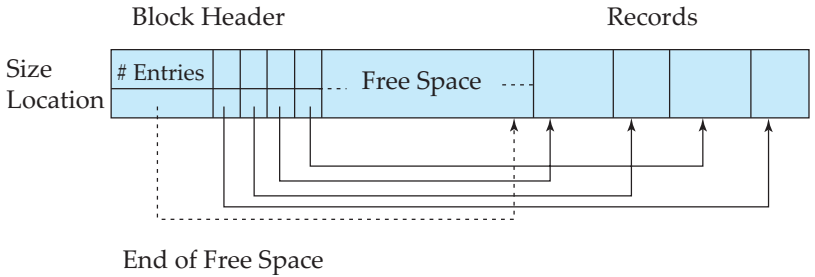
header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	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)

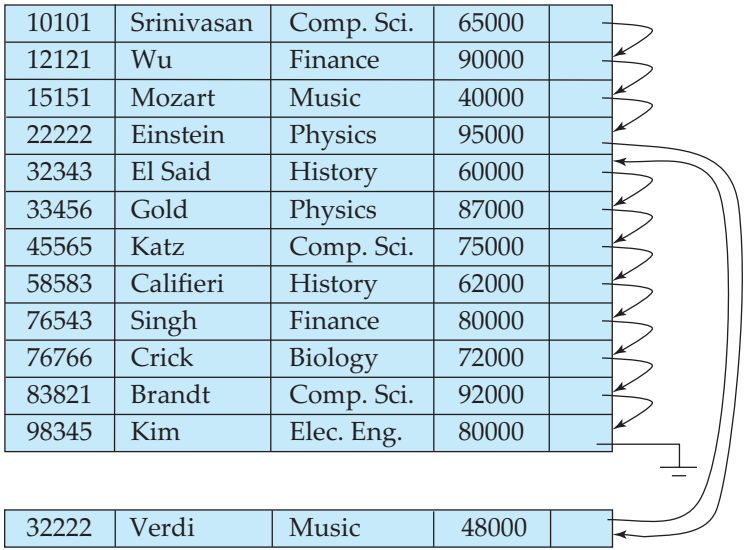
Sequential File Organization

- 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



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

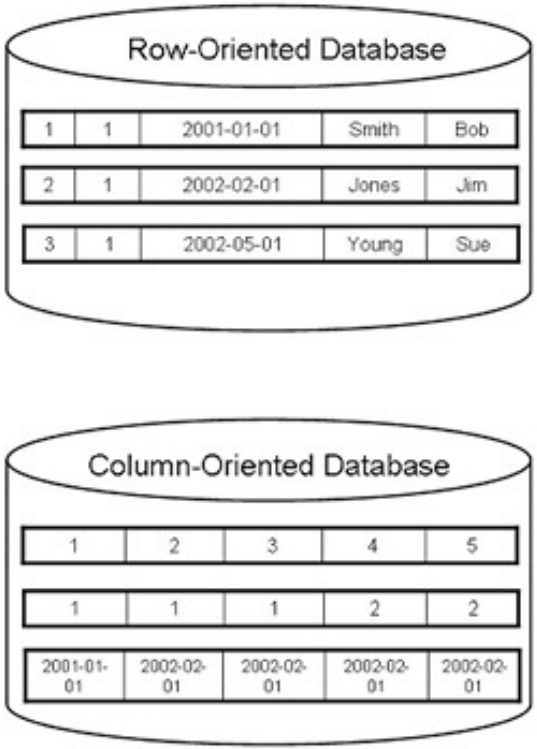
Columnar Representation

- 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

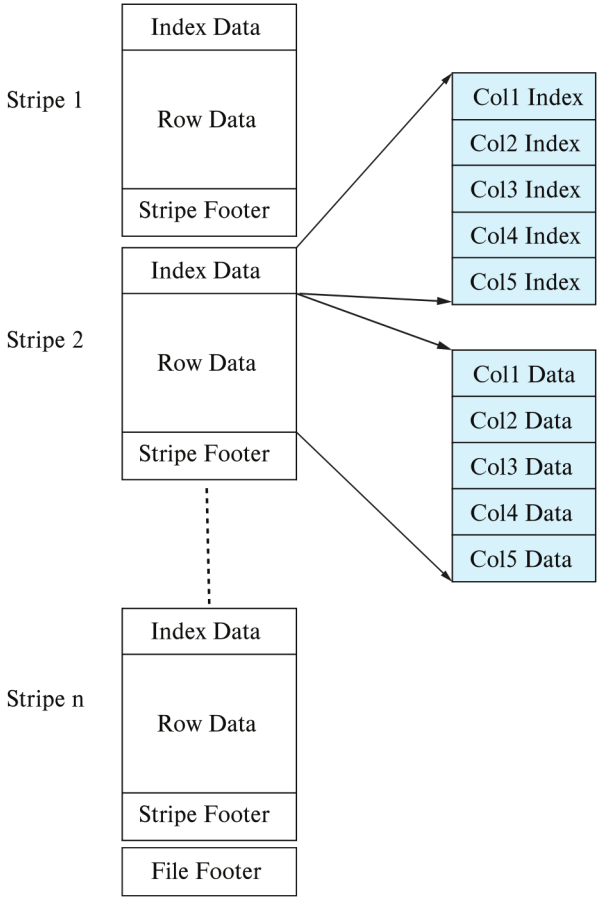
- Columnar and Row are both
  - Relational
  - Support SQL operations
- But differ in data storage
  - Row keeps row data together in blocks.
  - Columnar keys column data together in blocks.
- This determines performance for diffe types of query, e.g.
  - Columnar is extremely powerful for B
    - Aggregation 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

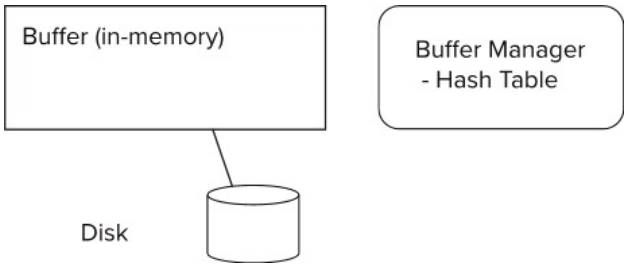
- ORC and Parquet: file formats with columnar storage inside file
- Very popular for big-data applications
- Orc file format shown on right:



Memory and Buffer Pools
Very Simple Overview

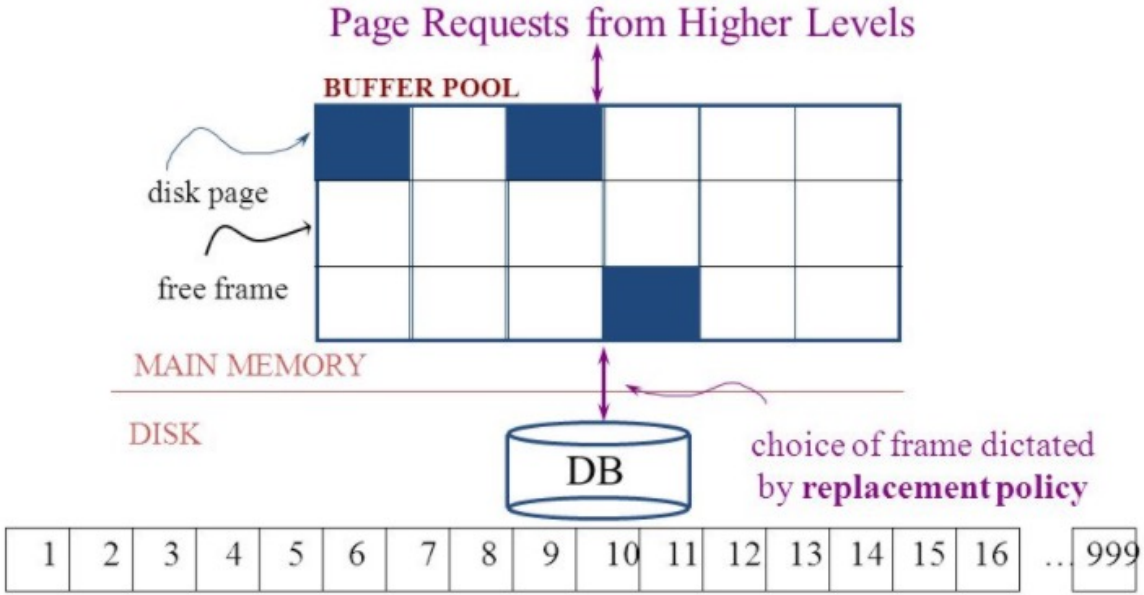
Storage Access

- 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.)

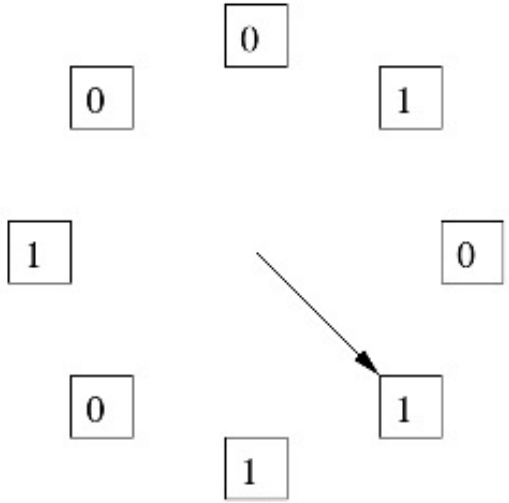
- **Toss-immediate** strategy – frees the space occupied by a block as soon as the final tuple of that block has been processed
- **Most recently used (MRU) strategy** – system must pin the block currently being processed. After the final tuple of that block has been processed, the block is unpinned, 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

Module III – NoSQL Continued

Reminder

One 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.

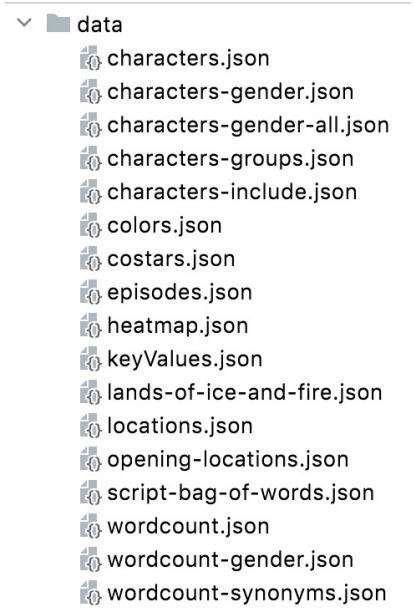
What is wrong with SQL/Relational?

- 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

Downloaded a Game-of-Thrones Dataset

- https://jeffreylancaster.github.io/game-of-thrones/
- The data is in a bunch of JSON (document) files.
- Information about relationships between characters is scattered through several files.



characters.json

```
{
   "characterName":"Aegon Targaryen",
   "houseName":"Targaryen",
   "royal":true,
   "parents":[
      "Elia Martell",
      "Rhaegar Targaryen"
   ],
   "siblings":[
      "Rhaenys Targaryen",
      "Jon Snow"
   ],
   "killedBy":[
      "Gregor Clegane"
   ]
},
```

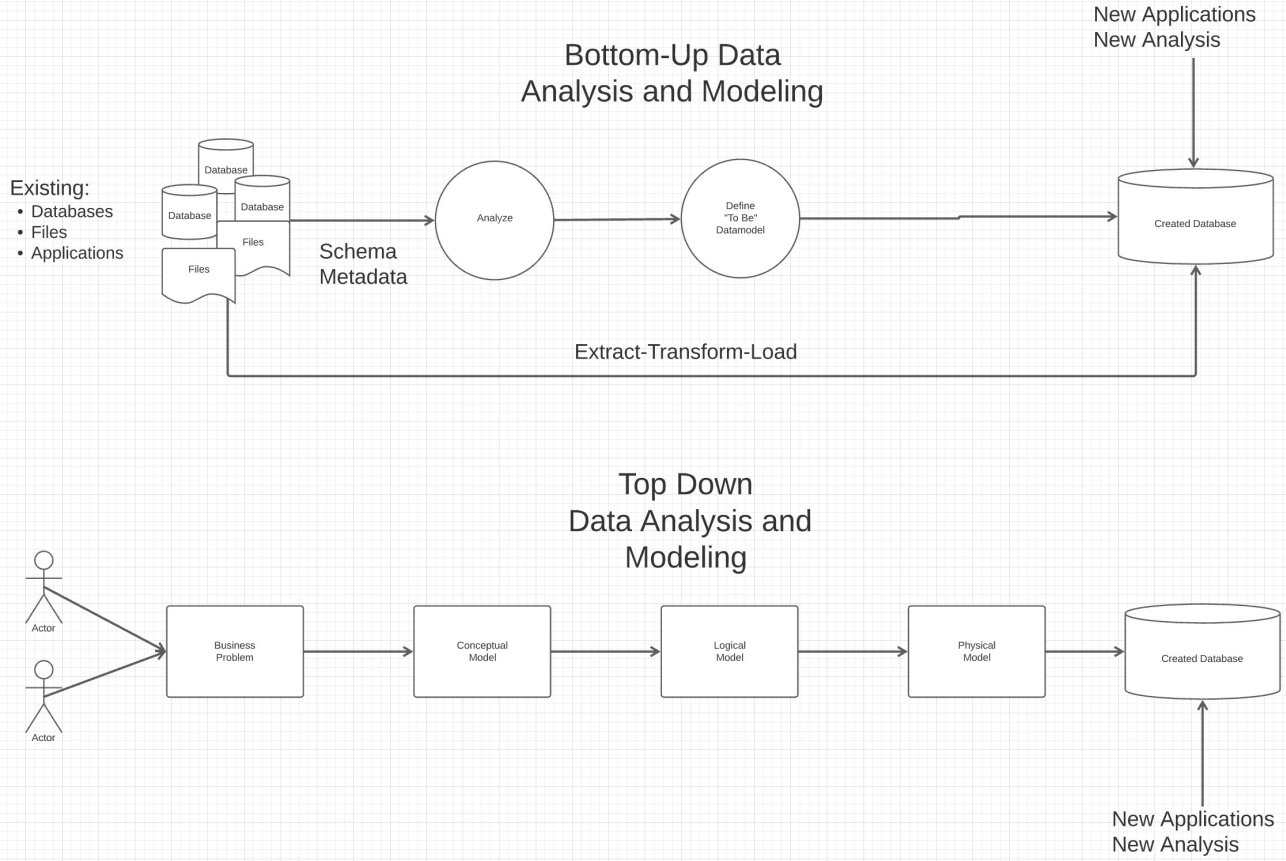
characters-groups.json

```
{
   "name":"Targaryen",
   "characters":[
      "Daenerys Targaryen",
      "Drogon",
      "Rhaegal",
      "Viserion",
      "Viserys Targaryen"
   ]
},
```

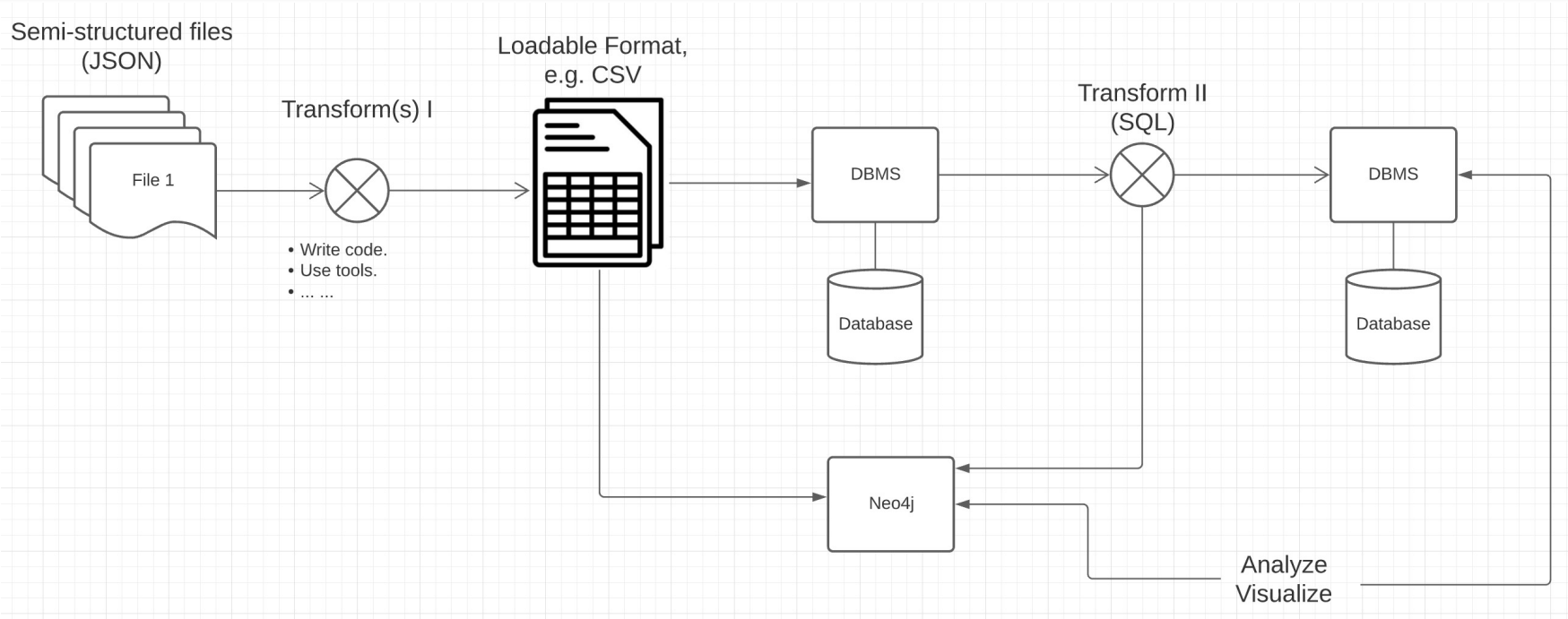
episodes.json

```
{
   "sceneStart":"0:05:58",
   "sceneEnd":"0:06:21",
   "location":"North of the Wall",
   "subLocation":"The Haunted Forest",
   "characters":[
      {"name":"Gared"},
      {
         "name":"Waymar Royce",
         "alive":false,
         "mannerOfDeath":"Back stab",
         "killedBy":[
            "White Walker"
         ]
      },
      {"name":"White Walker"}
   ]
},
```

Data Engineering and Data Modeling

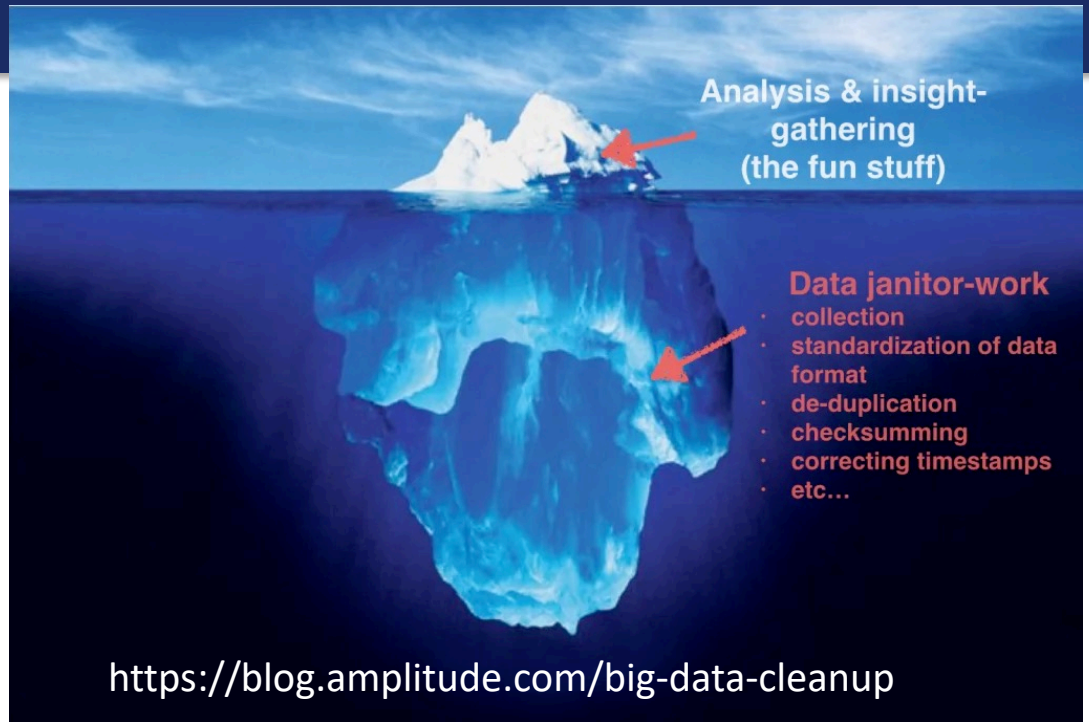


The Data is a Bit of a Mess → Cleanup Required



- For HW3, I will produce the CSV files.
- You will have to define desired schema and perform transformations.

Data Cleansing

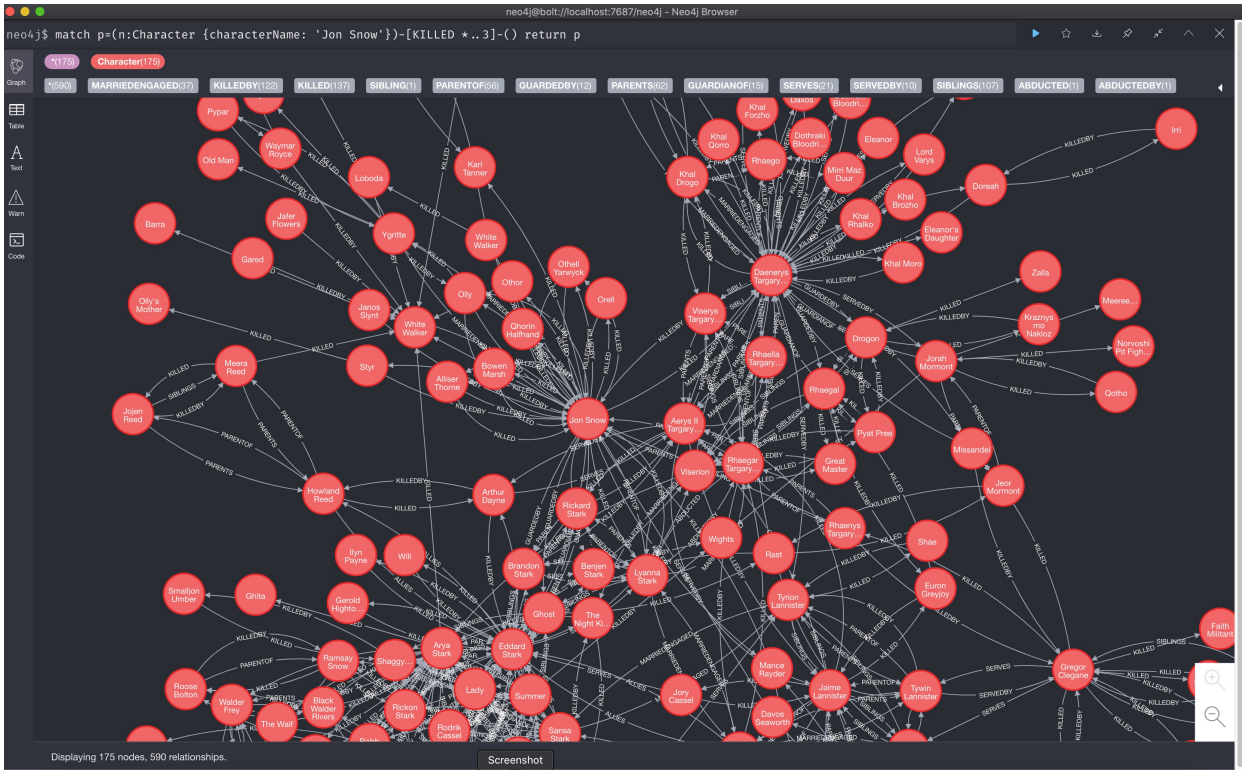


Database and data science classes **love to teach the fun stuff**
queries, data modeling, machine learning, spooky math, algorithms, ...

Switch to Notebook for Examples

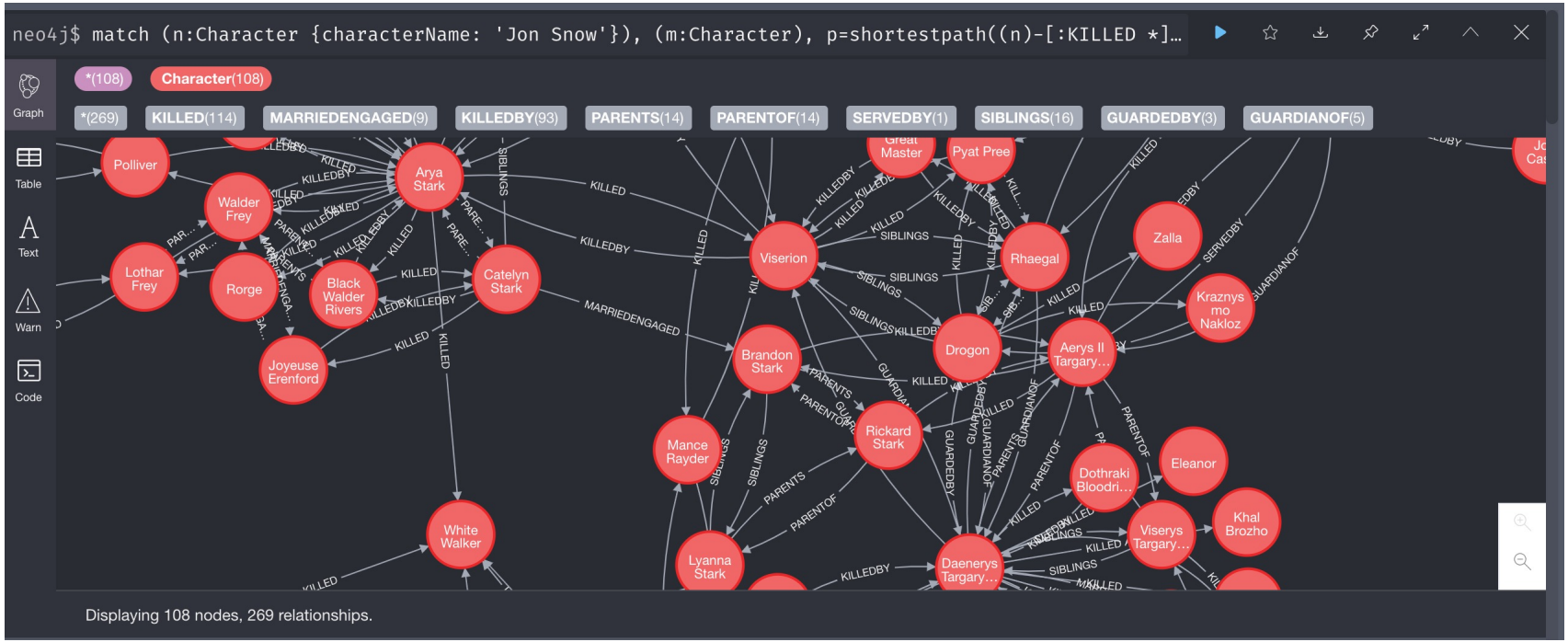
Path Query

```
match p=(n:Character {characterName: 'Jon Snow'})-[KILLED *..3]-()
return p
```



You can get Cycles/Duplicates → Can be Better

```
match (n:Character {characterName: 'Jon Snow'}), (m:Character),
p=shortestpath((n)-[:KILLED *]-(m)) return p
```



Document Databases and MongoDB

Disclaimers:

1. **We are only going to "touch" on MongoDB, like we did for Neo4j. NoSQL DBs are as complex as RDB and we spent several lectures on RDB.**
2. **Most of my document DB experience is with DynamoDB, not MongoDB.**

Some Context

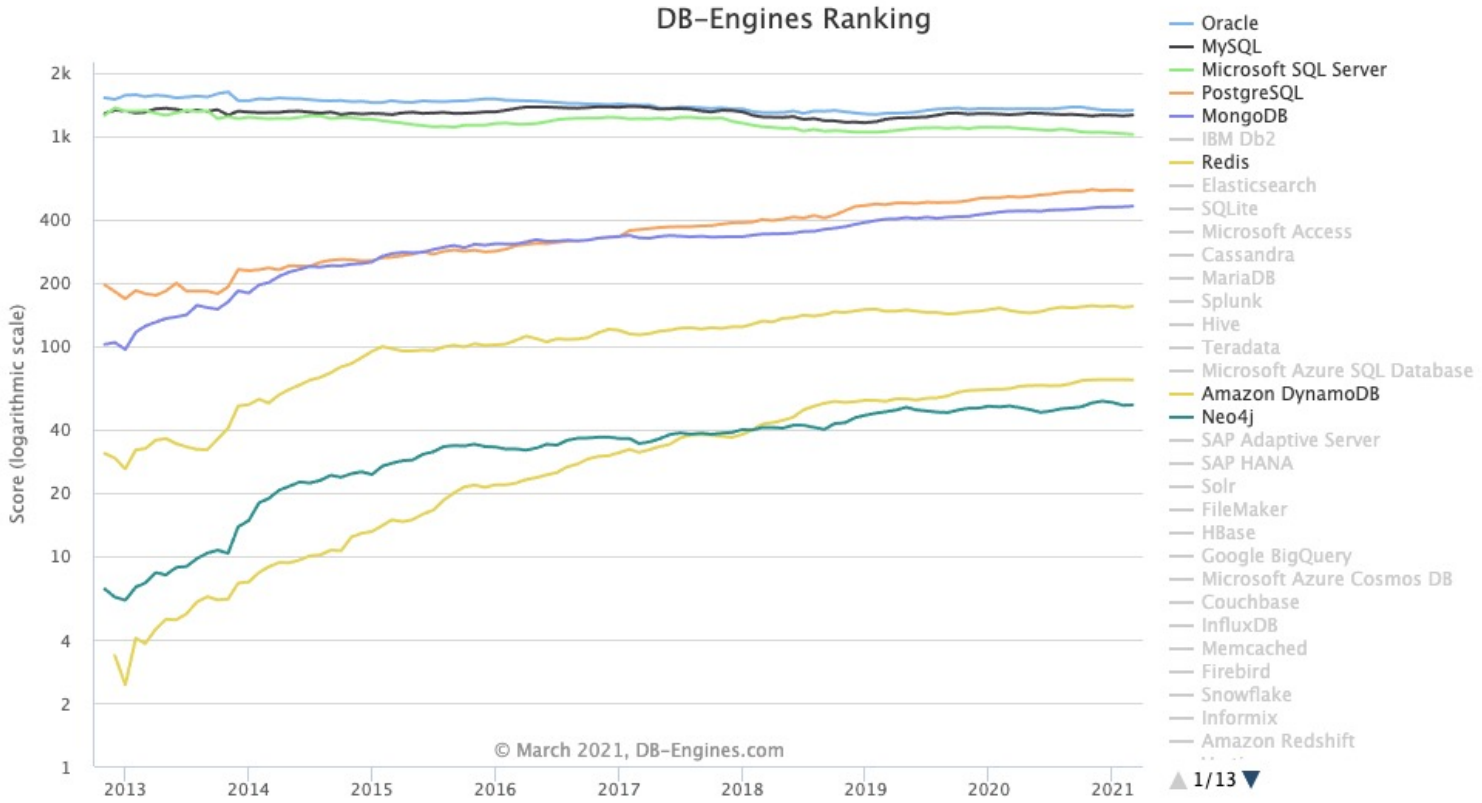
Database Adoption (https://db-engines.com/en/ranking)

364 systems in ranking, March 2021

Rank Mar 2021	Rank Feb 2021	Rank Mar 2020	DBMS	Database Model	Score Mar 2021	Score Feb 2021	Score Mar 2020
1.	1.	1.	Oracle	Relational, Multi-model	1321.73	+5.06	-18.91
2.	2.	2.	MySQL	Relational, Multi-model	1254.83	+11.46	-4.90
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	1015.30	-7.63	-82.55
4.	4.	4.	PostgreSQL	Relational, Multi-model	549.29	-1.67	+35.37
5.	5.	5.	MongoDB	Document, Multi-model	462.39	+3.44	+24.78
6.	6.	6.	IBM Db2	Relational, Multi-model	156.01	-1.60	-6.55
7.	7.	↑ 8.	Redis	Key-value, Multi-model	154.15	+1.58	+6.57
8.	8.	↓ 7.	Elasticsearch	Search engine, Multi-model	152.34	+1.34	+3.17
9.	9.	↑ 10.	SQLite	Relational	122.64	-0.53	+0.69
10.	↑ 11.	↓ 9.	Microsoft Access	Relational	118.14	+3.97	-7.00
11.	↓ 10.	11.	Cassandra	Wide column	113.63	-0.99	-7.32
12.	12.	↑ 13.	MariaDB	Relational, Multi-model	94.45	+0.56	+6.10
13.	13.	↓ 12.	Splunk	Search engine	86.93	-1.61	-1.59
14.	14.	14.	Hive	Relational	76.04	+3.72	-9.34
15.	↑ 16.	15.	Teradata	Relational, Multi-model	71.43	+0.53	-6.41
16.	↓ 15.	↑ 23.	Microsoft Azure SQL Database	Relational, Multi-model	70.88	-0.41	+35.44
17.	17.	↓ 16.	Amazon DynamoDB	Multi-model	68.89	-0.25	+6.38
18.	↑ 19.	↑ 21.	Neo4j	Graph, Multi-model	52.32	+0.16	+0.54
19.	↓ 18.	↑ 20.	SAP Adaptive Server	Relational, Multi-model	52.17	-0.07	-0.59
20.	↑ 21.	↓ 18.	SAP HANA	Relational, Multi-model	51.00	+0.77	-3.27

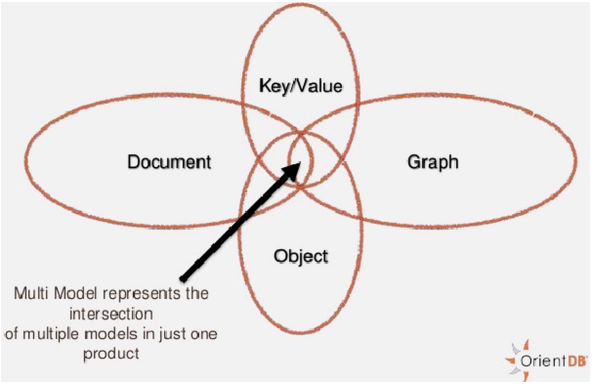
Multi-model mean
that the DB engine sup
more than one cor
approach,
e.g. RDB and Docume

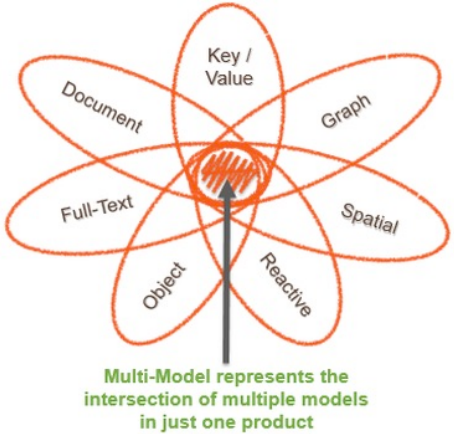
Database Adoption (https://db-engines.com/en/ranking)



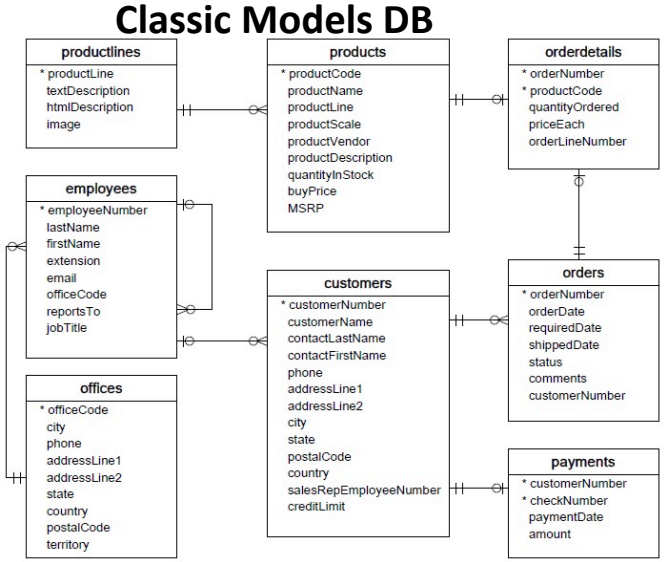
Multi-model Database

- "In the field of database design, a multi-model database is a database management system designed to support multiple data models against a single, integrated backend. In contrast, most database management systems are organized around a single data model that determines how data can be organized, stored, and manipulated.[1] Document, graph, relational, and key-value models are examples of data models that may be supported by a multi-model database." (https://en.wikipedia.org/wiki/Multi-model_database)
- I use Azure Cosmos DB in one of my current projects, which
  - Data models: SQL on JSON, Document (MongoDB), Column-family, Key-Value
  - Has various forms of transactions, stored procedures, sharding, ...





Motivation: Some Datamodels are Nested



The way people think about orders.

- classicmodels is a relational database (https://www.mysqltutorial.org/mysql-sample-database.aspx/)
- Relational is good for many, many things and is a great model, but some user perspectives expose data differently. In this case: table versus document (form)

MongoDB Overview

Copied from online sources.
If you get to use Google/StackOverflow for code,
I get to use Google/SlideShare for presentations.

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.

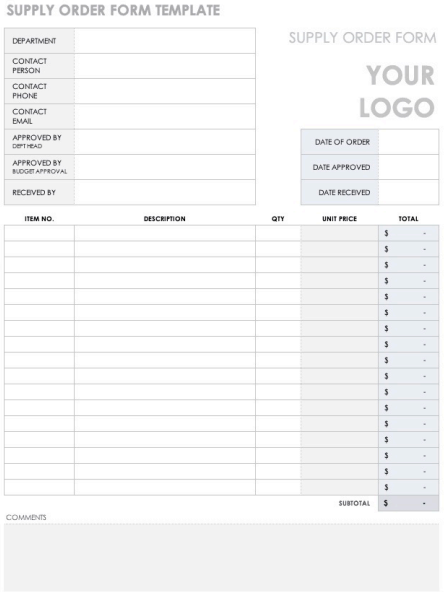
Datamodel Impedance Match/Mismatch

Document Format:


```
{
  "orderNumber": 10100,
  "customerNumber": 363,
  "orderDate": "2003-01-06",
  "requiredDate": "2003-01-13",
  "shippedDate": "2003-01-10",
  "status": "Shipped",
  "orderDetails": [
   {
    "orderLineNumber": 1,
    "productCode": "S24_3969",
    "quantityOrdered": 49,
    "priceEach": "35.29"
   },
   {
    "orderLineNumber": 2,
    "productCode": "S18_2248",
    "quantityOrdered": 50,
    "priceEach": "55.09"
   },
   {
    "orderLineNumber": 3,
    "productCode": "S18_1749",
    "quantityOrdered": 30,
    "priceEach": "136.00"
   },
   {
    "orderLineNumber": 4,
    "productCode": "S18_4409",
    "quantityOrdered": 22,
    "priceEach": "75.46"
   }
  ]
}
```


orderNumber	customerNumber	orderDate	requiredDate	shippedDate	status	productCode	quantityOrder...	priceEach	orderLineNumber
10100	363	2003-01-06	2003-01-13	2003-01-10	Shipped	S24_3969	49	35.29	1
10100	363	2003-01-06	2003-01-13	2003-01-10	Shipped	S18_2248	50	55.09	2
10100	363	2003-01-06	2003-01-13	2003-01-10	Shipped	S18_1749	30	136.00	3
10100	363	2003-01-06	2003-01-13	2003-01-10	Shipped	S18_4409	22	75.46	4

More Natural Mapping



Impedance mismatch is the term used to refer to the problems that occurs due to differences between the database model and the programming language model. The practical relational model has 3 components these are:

1. Attributes and their data types
2. Tuples
3. Tables/collections/sets
4. Relationships

(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	Variables
$unwind	$avg		
$group	$push		$let
$geoNear	$sum		$map
$text			
$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 more in subsequent lectures, homework, … …
- You will have to install MongoDB and Compass for HW3, HW4 and final exam.

Switch to Notebook

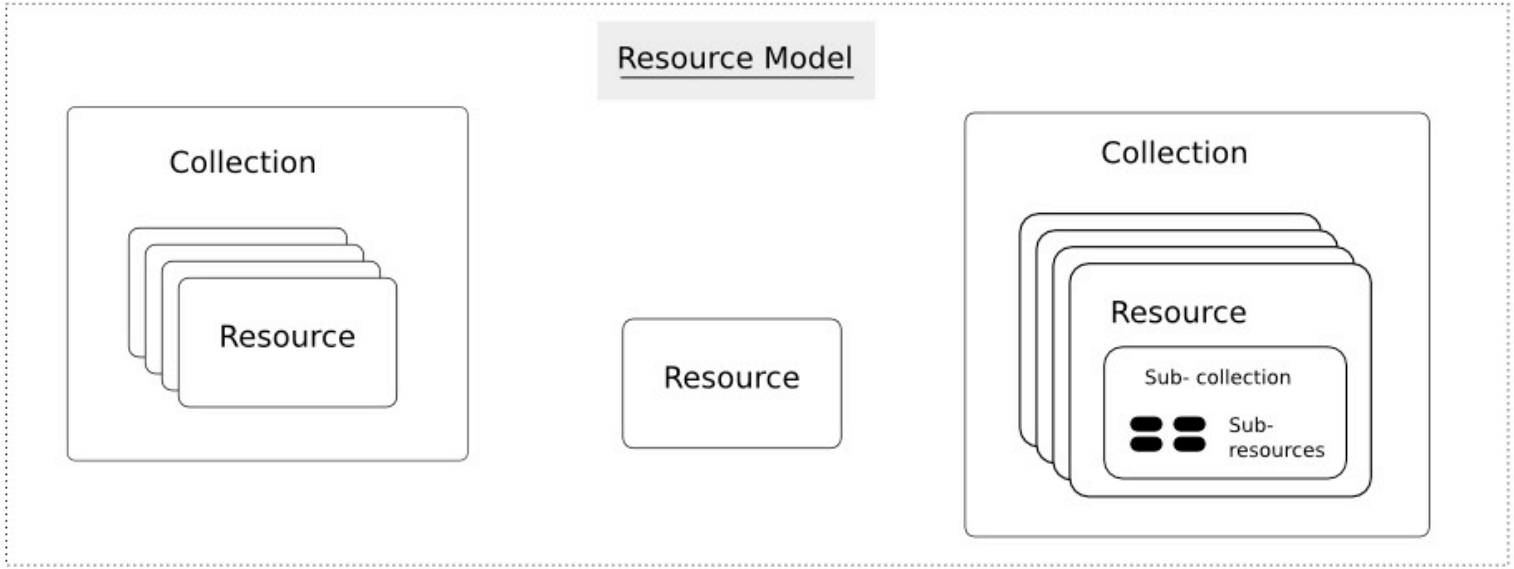
Data Models and REST

Data Modeling Concepts and REST

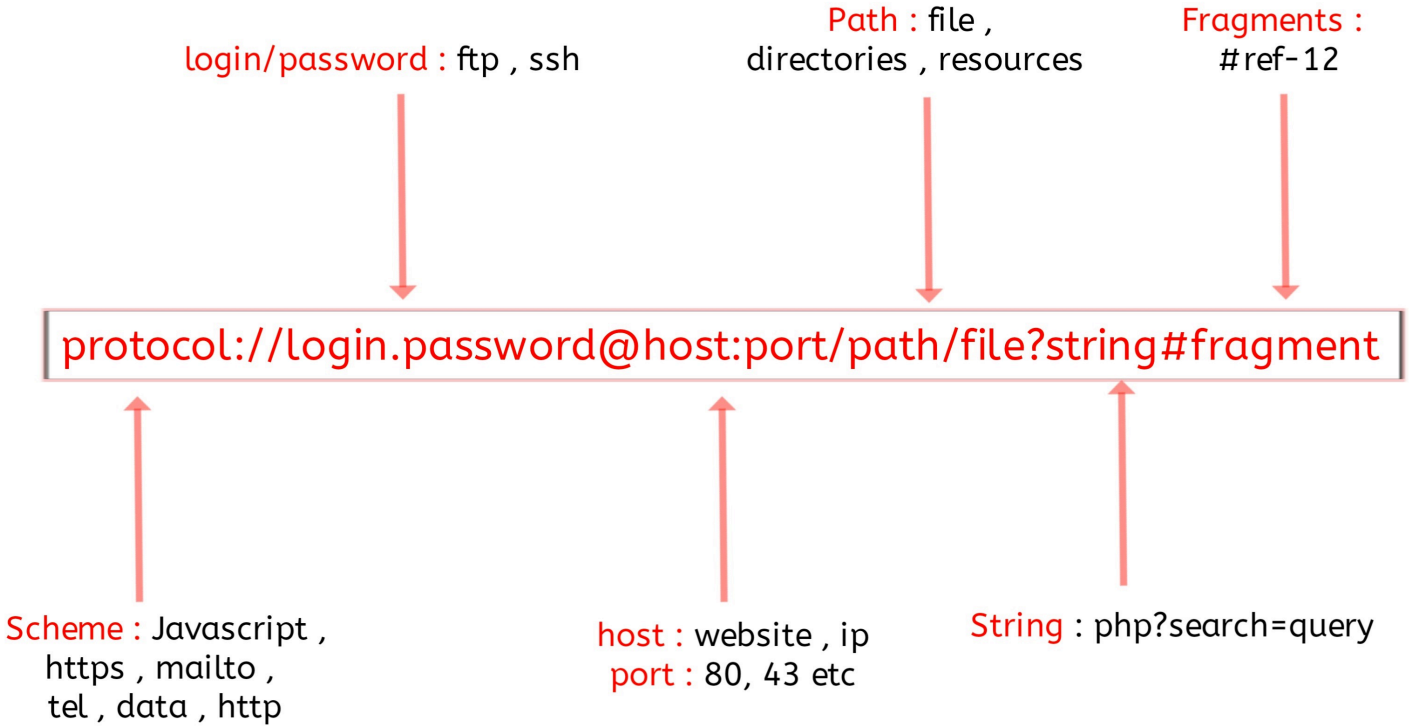
Almost any data model has the same core concepts:

- Types and instances:
  - Entity Type: A definition of a type of thing with properties and relationships.
  - Entity Instance: A specific instantiation of the Entity Type
  - Entity Set Instance: An Entity Type that:
    - Has properties and relationships like any entity, but ...
    - Has at least one *special relationship* – ***contains.***
- Operations, minimally CRUD, that manipulate entity types and instances:
  - Create
  - Retrieve
  - Update
  - Delete
  - Reference/Identify/... ...

REST and Resources



URLs



URL Mappings

- Some URLs: This gets you to the database service (program)
  - http://127.0.0.1:5001/api
  - mysql+pymysql://dbuser:dbuserdbuser@localhost
  - mongodb://mongodb0.example.com:27017
  - neo4j://graph.example.com:7687
- You still have to get into a database within the service:
  - SQL: use lahmansbaseballdb
  - MongoDB: db.lahmansbaseballdb
  - HTTP://127.0.0.1:5001/api/lahmansbaseballdb
  - … …
- And then into things inside of things inside of things … In the database.

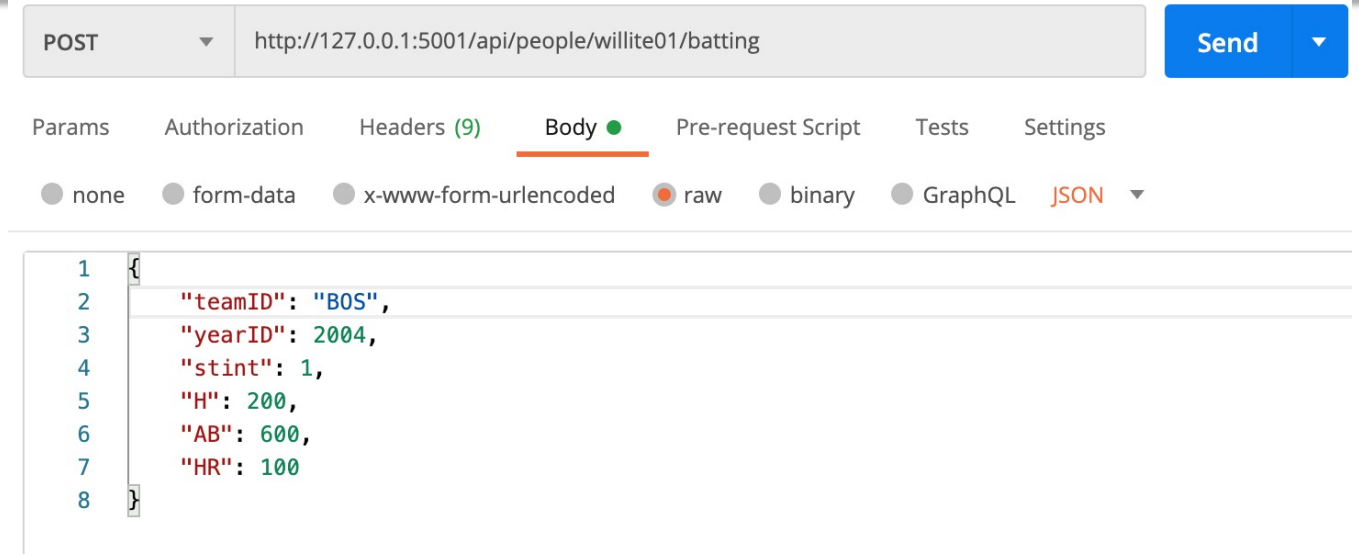
Simplistic, Conceptual Mapping (Examples)

REST Method	Resource Path	Relational Operation	DB Resource
DELETE	/people	DROP TABLE	people table
POST	/people	INSERT INTO PEOPLE (...) VALUES(...)	people table people row
GET	/people/21	SHOW KEYS FROM people ...; SELECT * FROM people WHERE playerID= 21	people row
GET	/people/21/batting	SELECT batting.* FROM people JOIN batting USING(playerID) WHERE playerID=21	
GET	/people/21/batting/2004_1	SELECT batting.* FROM people JOIN batting USING(playerID) WHERE playerID=21 AND yearID=2004 AND stint=1	

PUT, DELETE, UPDATE

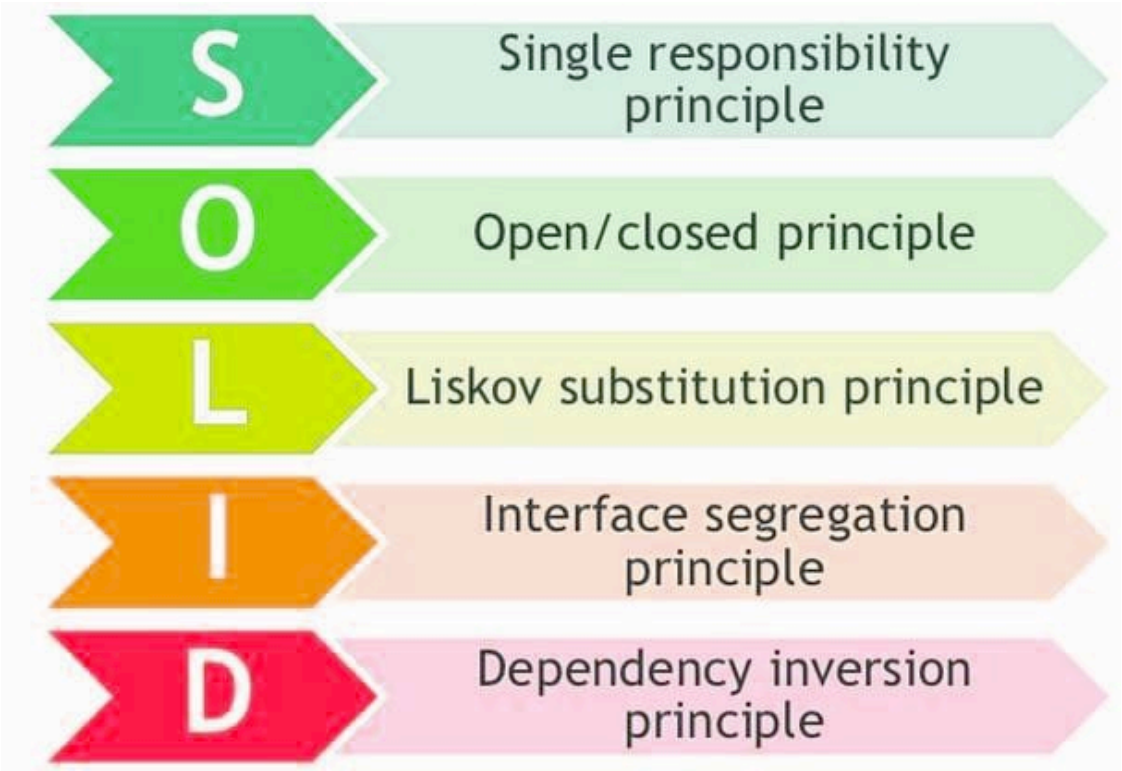
- /people?
  - POST (INSERT)
  - GET (SELECT ... WHERE ...)
- /people/21
  - WHERE peopleID=21
  - DELETE → DELETE WHERE
  - PUT → UPDATE SET ..... WHERE
  - GET SELECT ... WHERE

Simplistic, Conceptual Mapping (Examples)

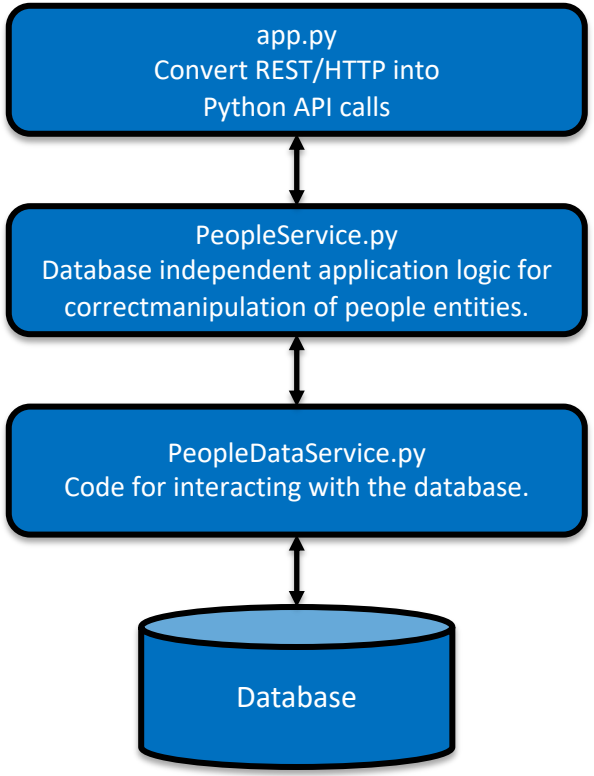



```
INSERT INTO
    batting(playerID, teamID, yearID, stint, H, AB, HR)
    VALUES ("willite01", "BOS", 2004, 1, 200, 600, 100)
```

SOLID (SW) Design Principle



Single Responsibility

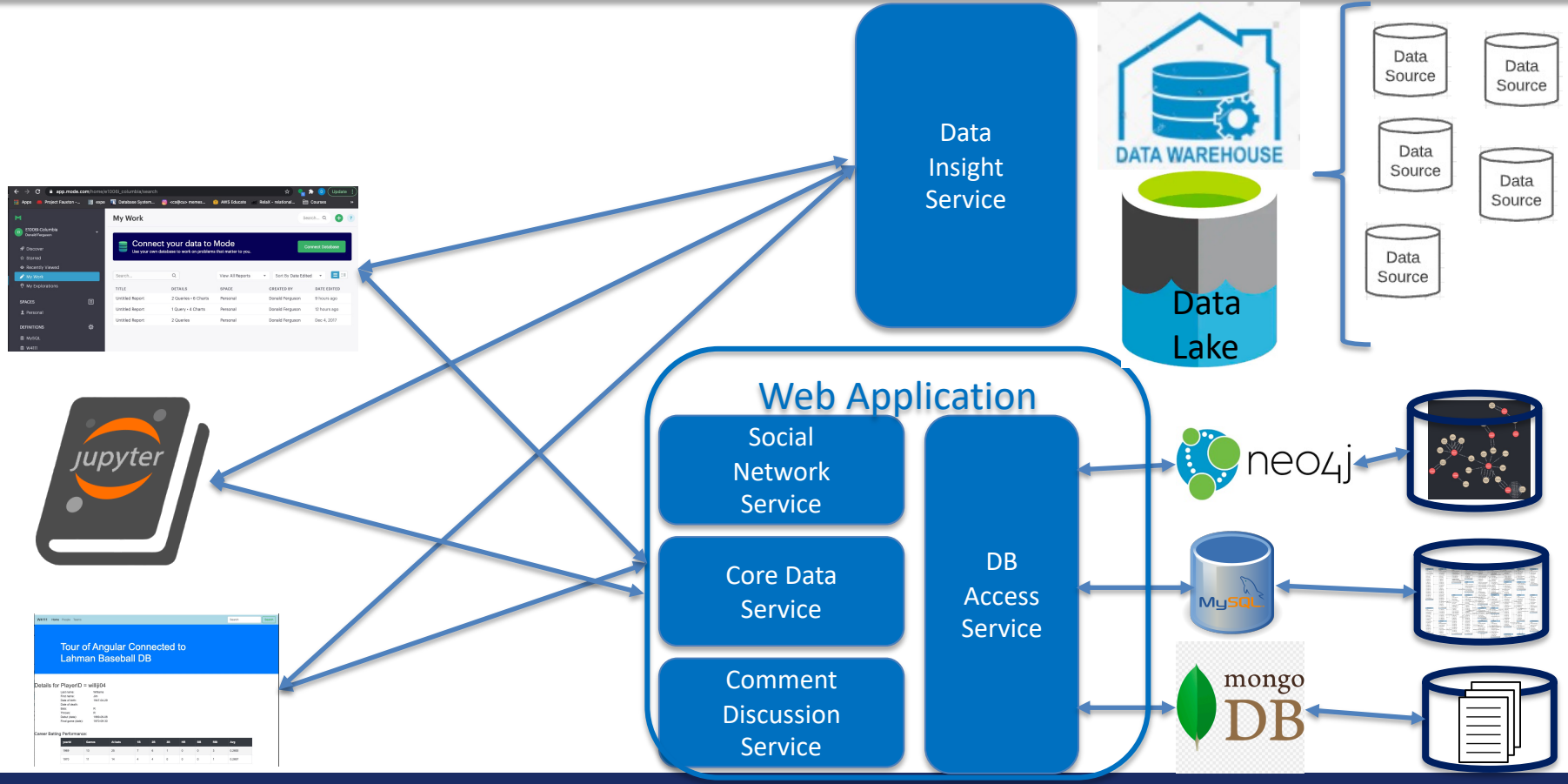


HW3 (Project)
To Be Refined and Simplified!

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)



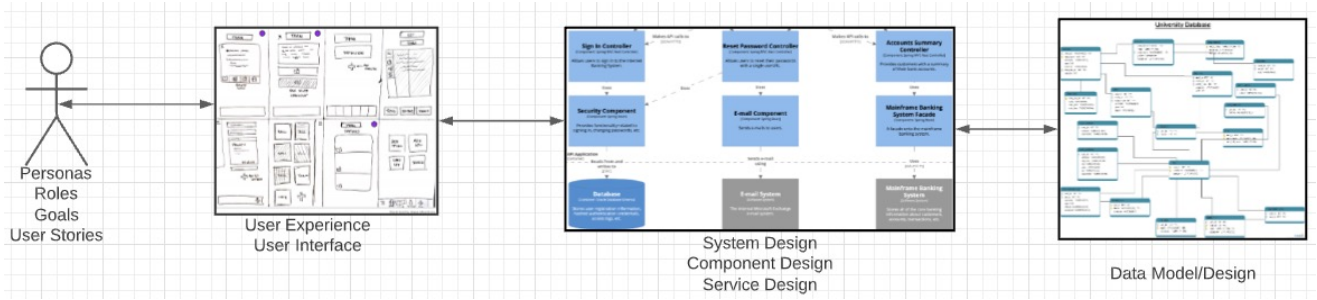
Target Application/Project – Reminder (Lecture 1)

- 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)

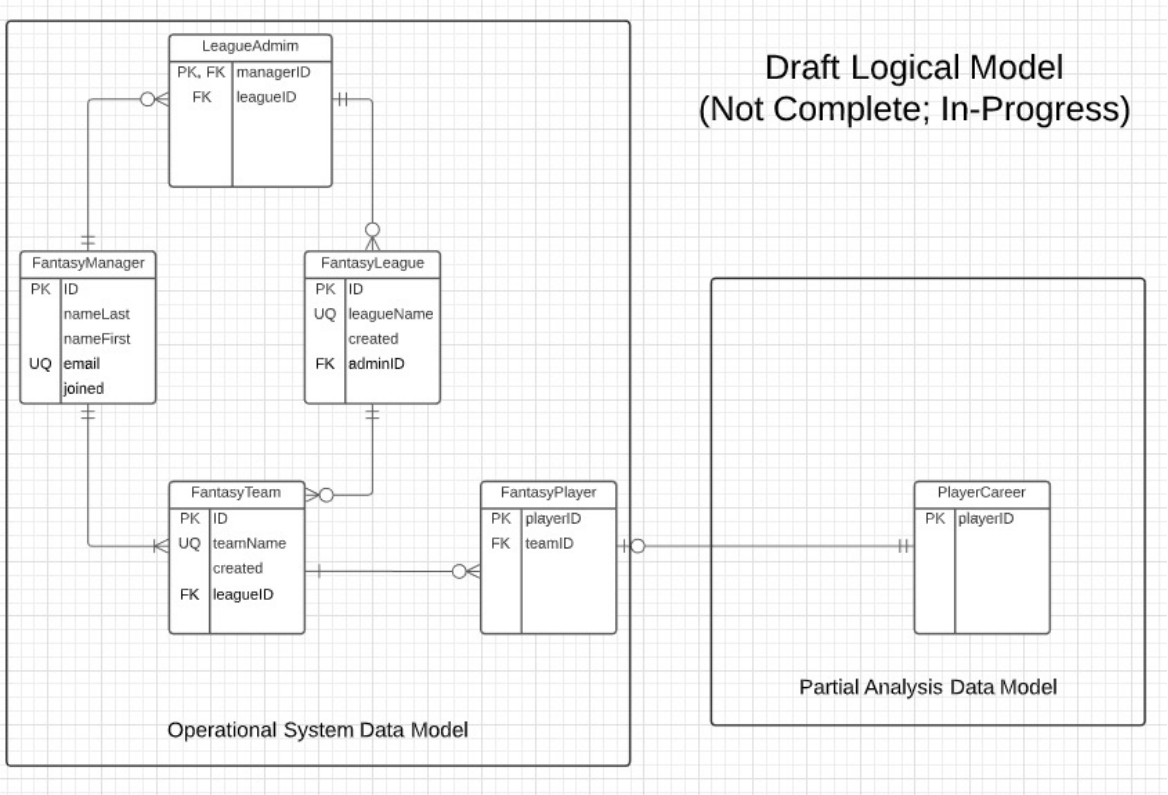
- We must build a system that supports … … operations in a … …
- We will design, develop, test and deploy the system iteratively and continuously.
- There are four core domains.



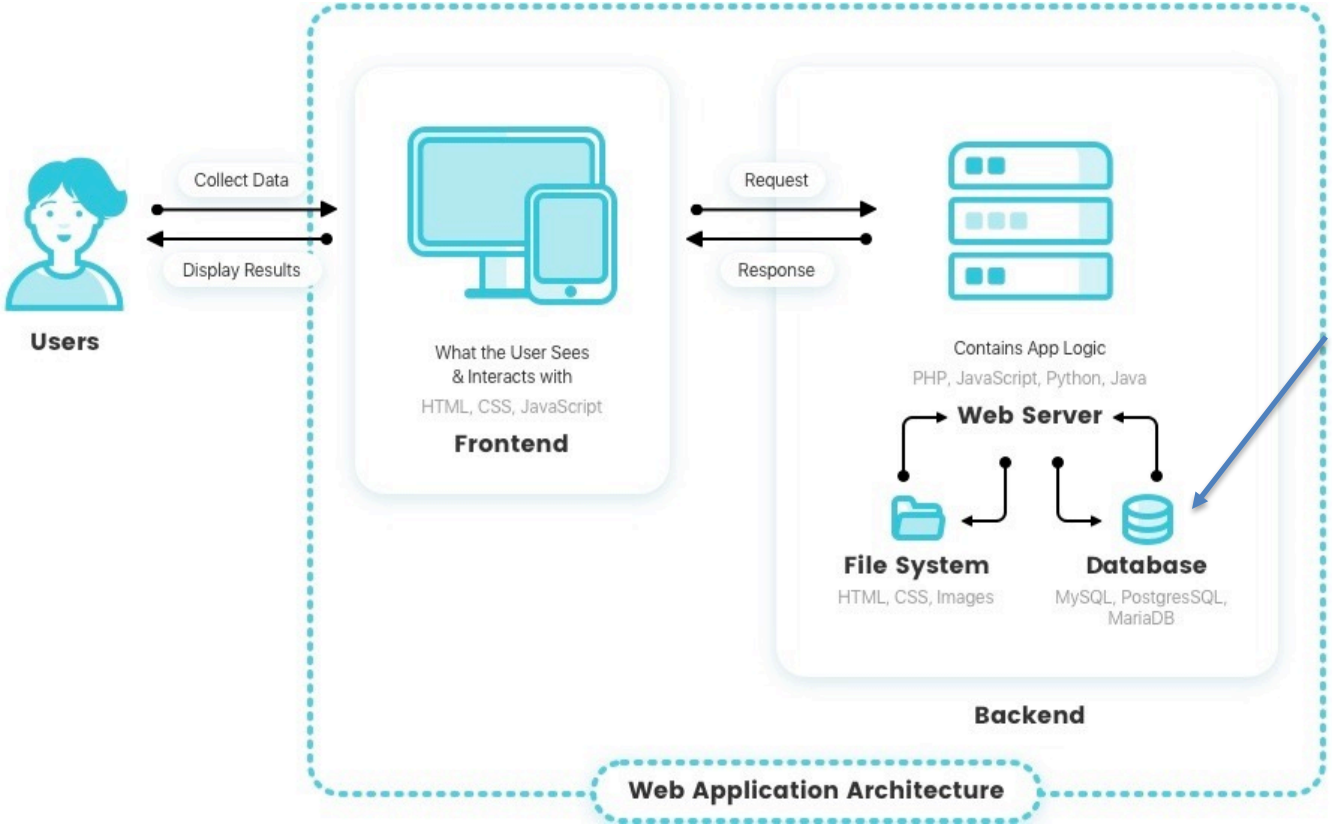
- In this course,
- We focus on the data dimension.
- We will get some insight into the other dimensions.

- 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



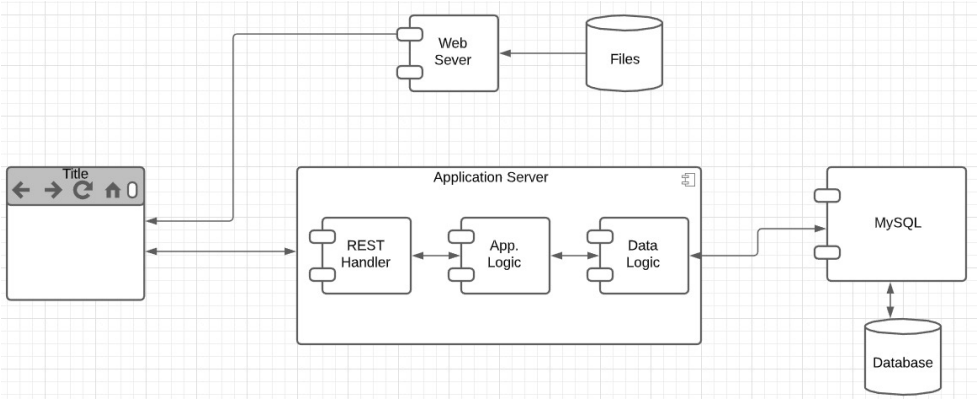
Web Application – Operational System



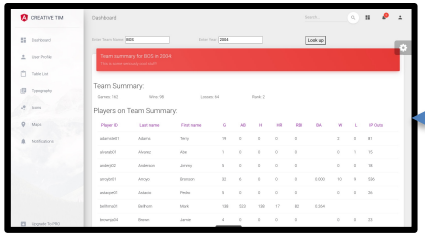
Lahman's Baseball DB
Fantasy Baseball DB

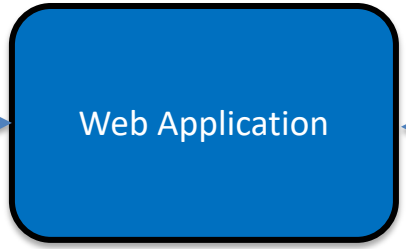
User Story

- “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.

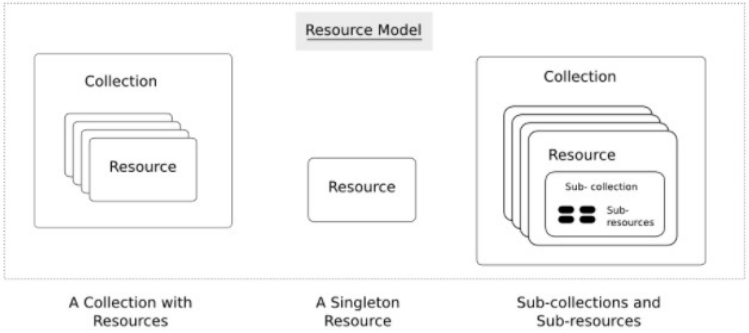
Tracks will build:

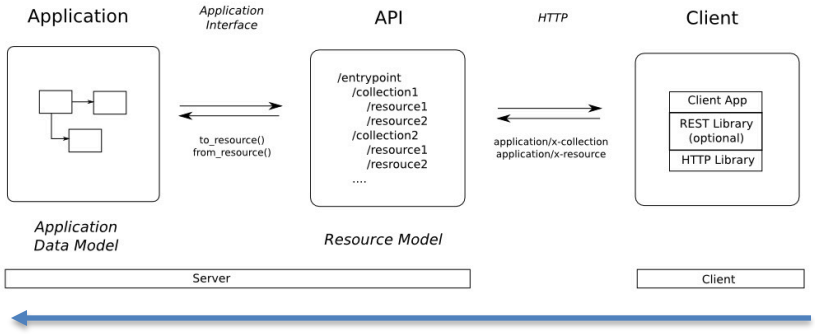
- Programming: Mix of CRUD and dashboard.
- Non-programming: Dashboards, Data transformation.

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 ... ...

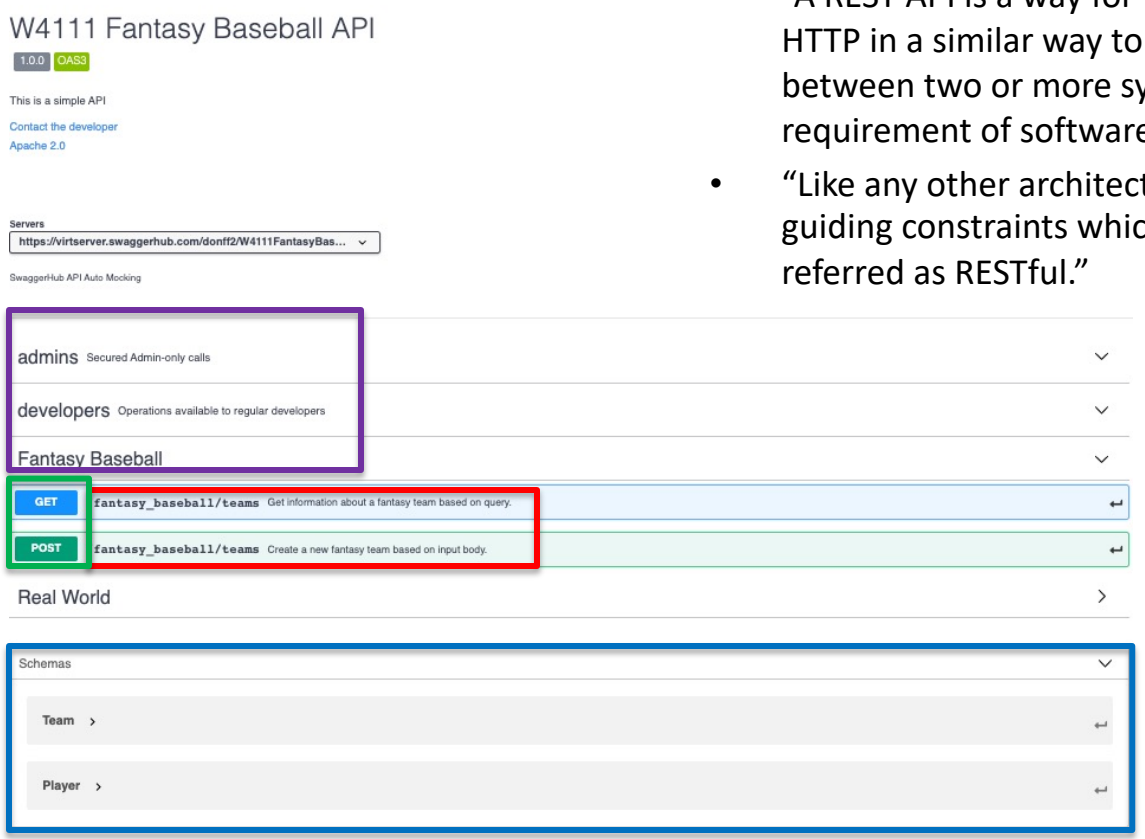
CRUD (https://en.wikipedia.org/wiki/Create,_read,_update_and_delete)

- 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

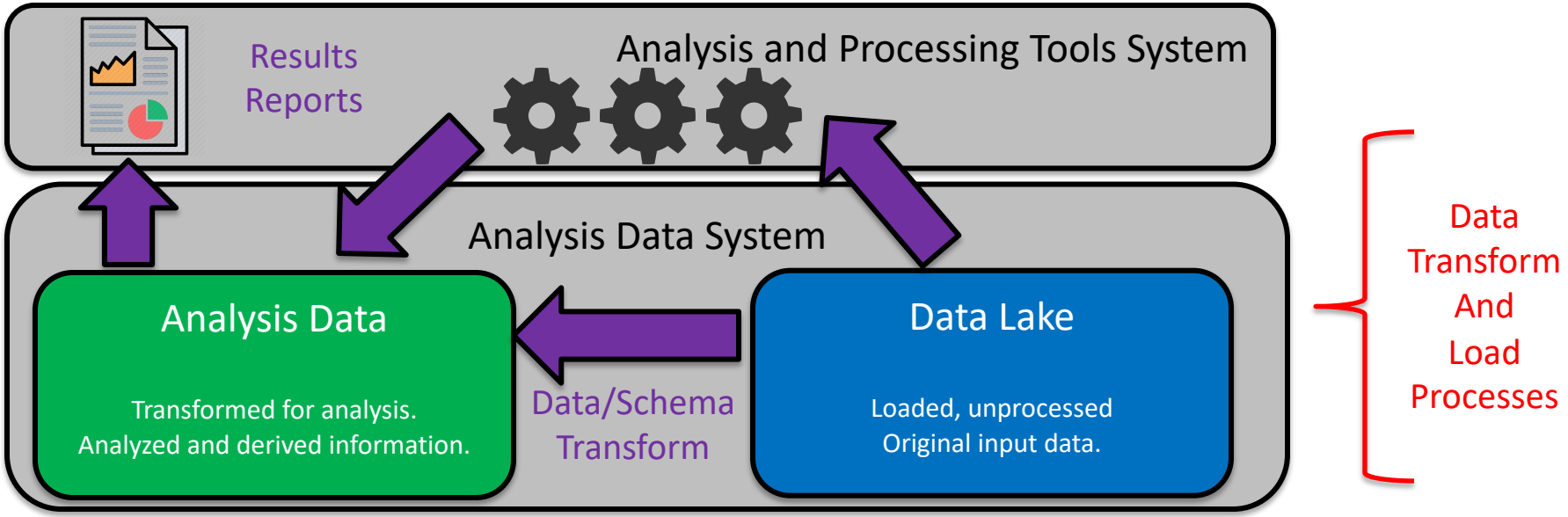
- API: https://app.swaggerhub.com/apis/donff2/W4111FantasyBaseball/1.0.0#/
- Starter API project:
  - App (Postman) http://127.0.0.1:5000/api/fantasy_team/BOS20011
  - Project: ~/Dropbox/Columbia/W4111_S21_New/W4111S21/Projects/FantasyBaseball

- 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)



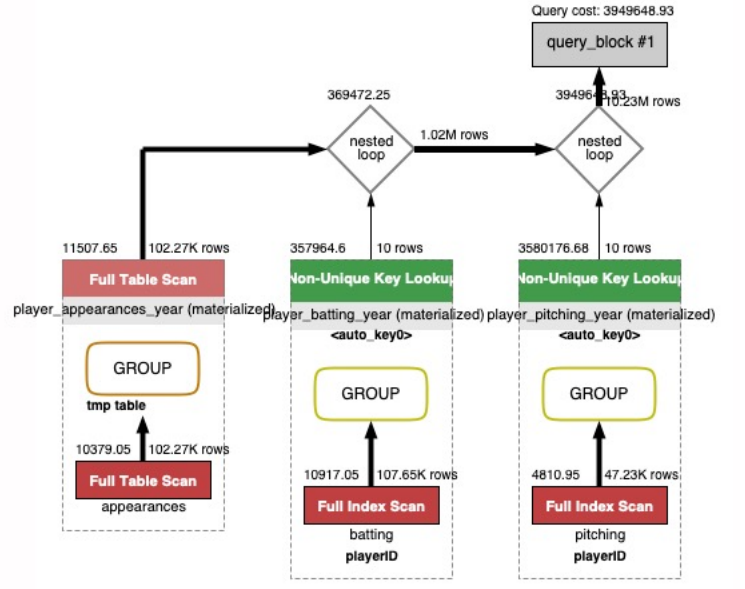
Data Lake to Analysis Data

- 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.

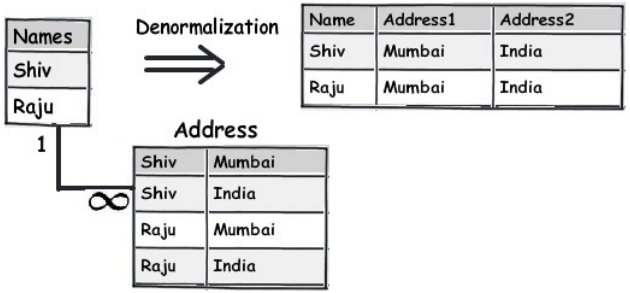
Task 1 – Player, Year Summary

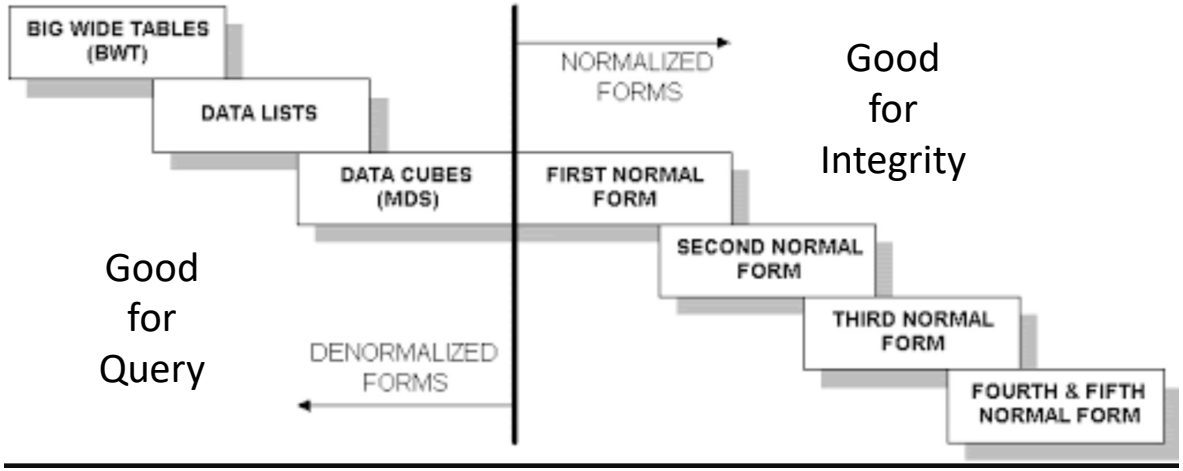
```
with player_basic
  as (
    select playerID, nameLast, nameFirst, bats, throws from people),
   player_appearances_year as
   (
     select
       playerid, yearid, sum(g_all) as app_g_all, sum(gs) as app_g_s, sum(g_defense) as app_g_defense,
       sum(g_p) as app_g_p, sum(g_c) as app_g_c, sum(g_1b) 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.

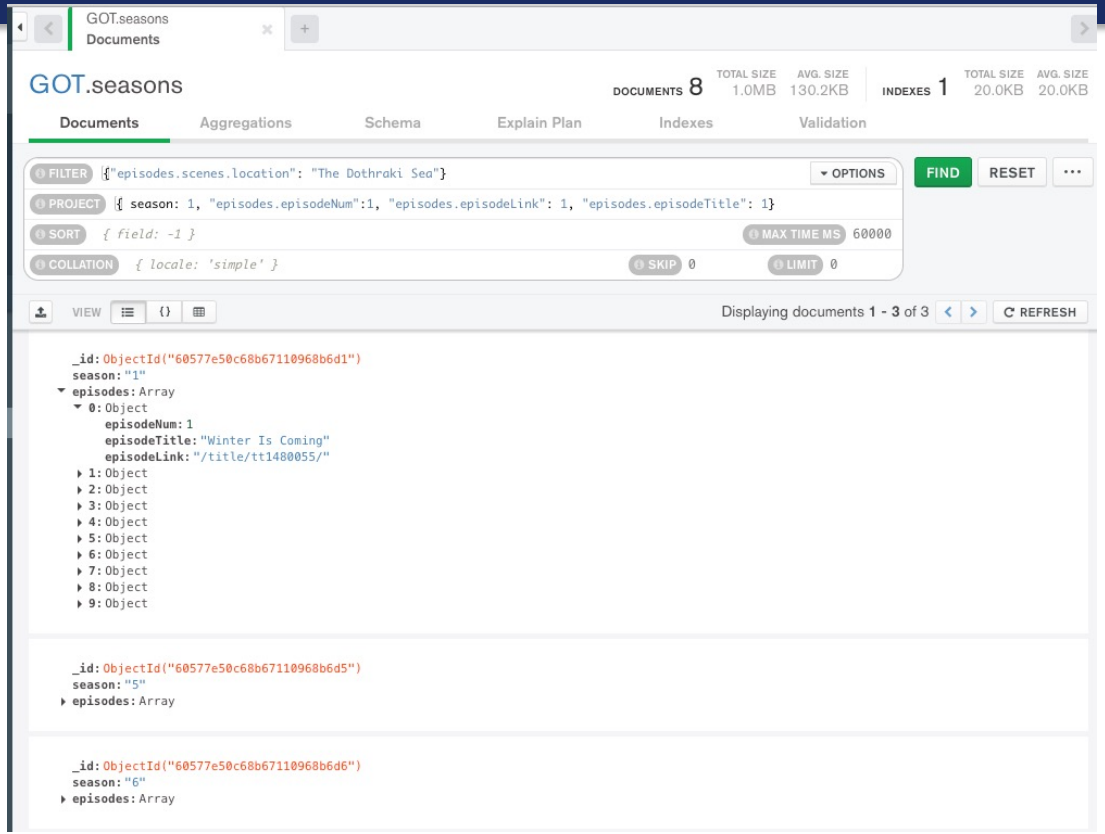
HW3, HW4 – The Project

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

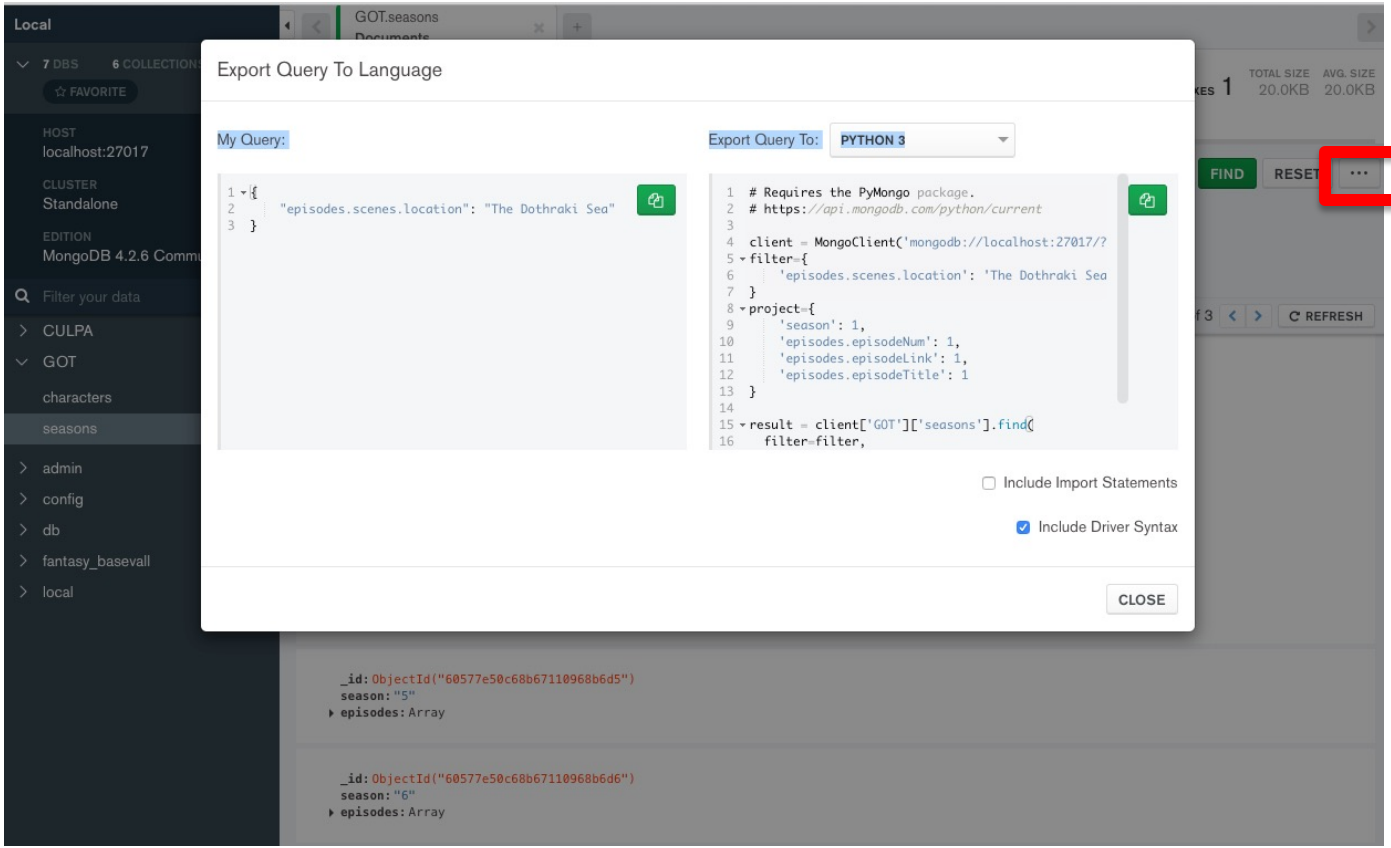
Stay Tuned.

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?



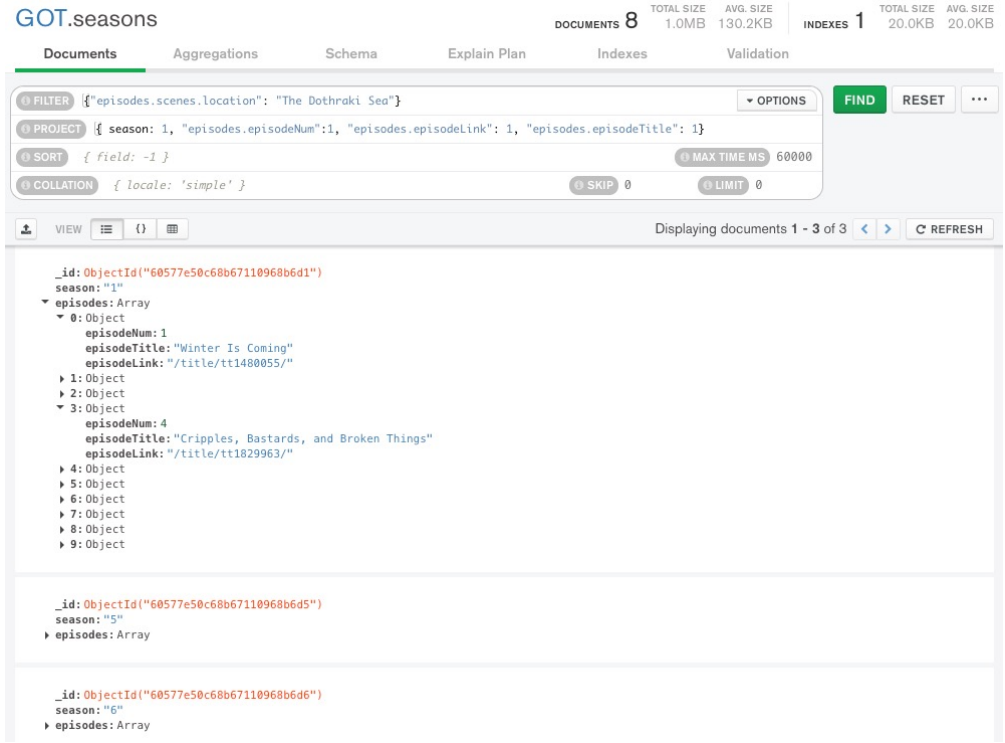
Generate Code



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

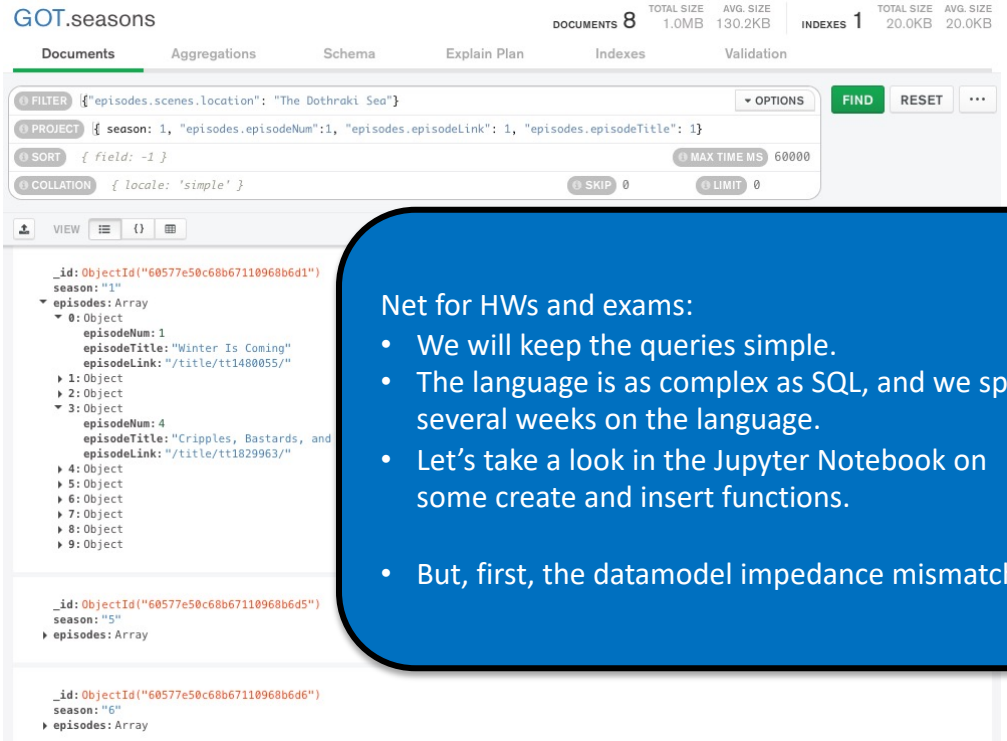
- Switch to Notebook

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:

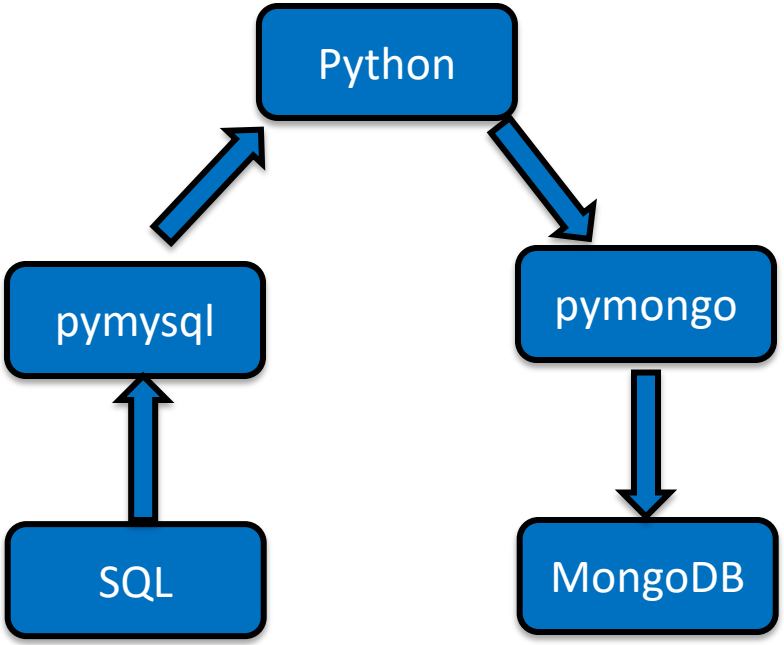
- We will keep the queries simple.
- The language is as complex as SQL, and we spent several weeks on the language.
- Let's take a look in the Jupyter Notebook on some create and insert functions.
- But, first, the datamodel impedance mismatch concept.

- 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.



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.