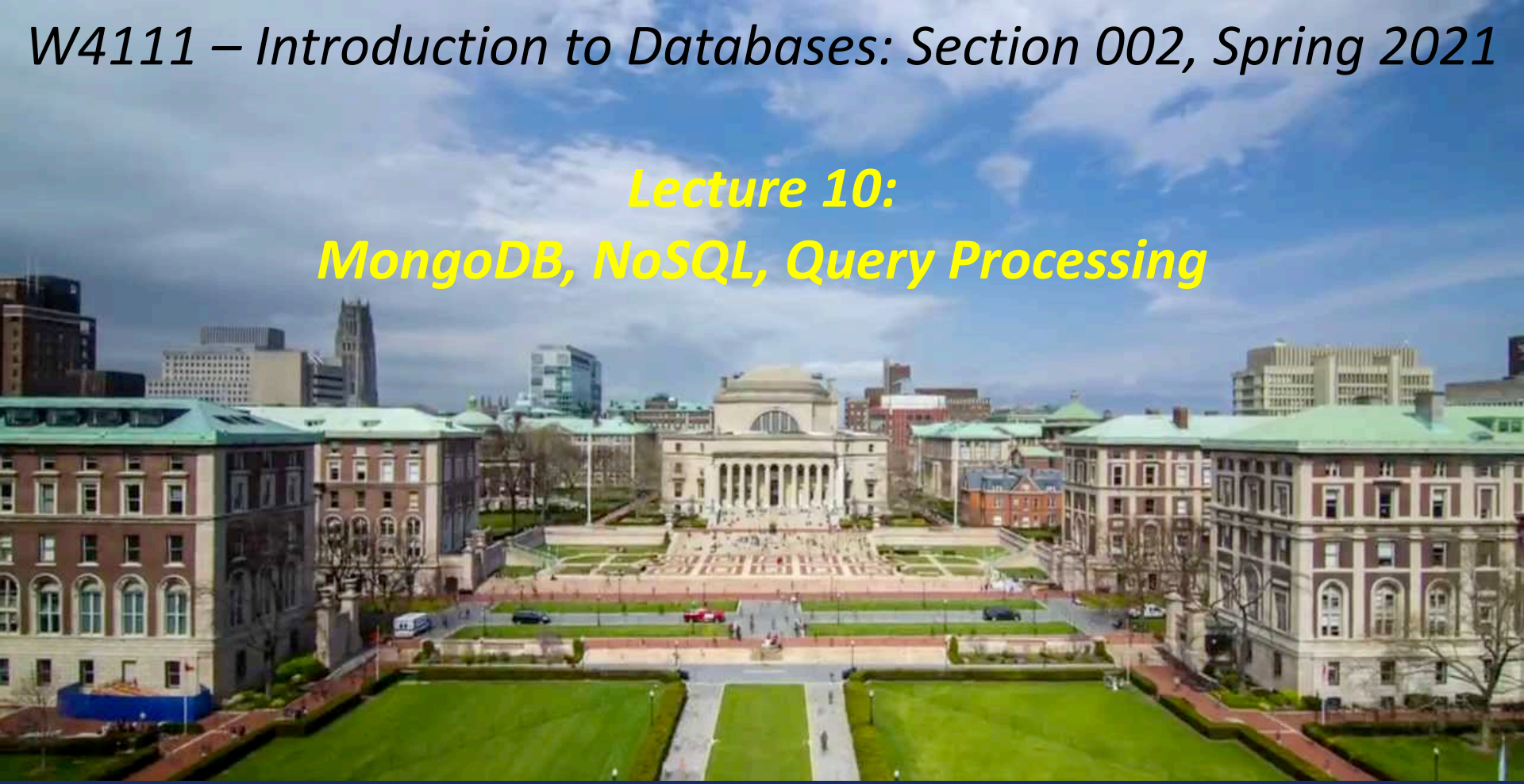
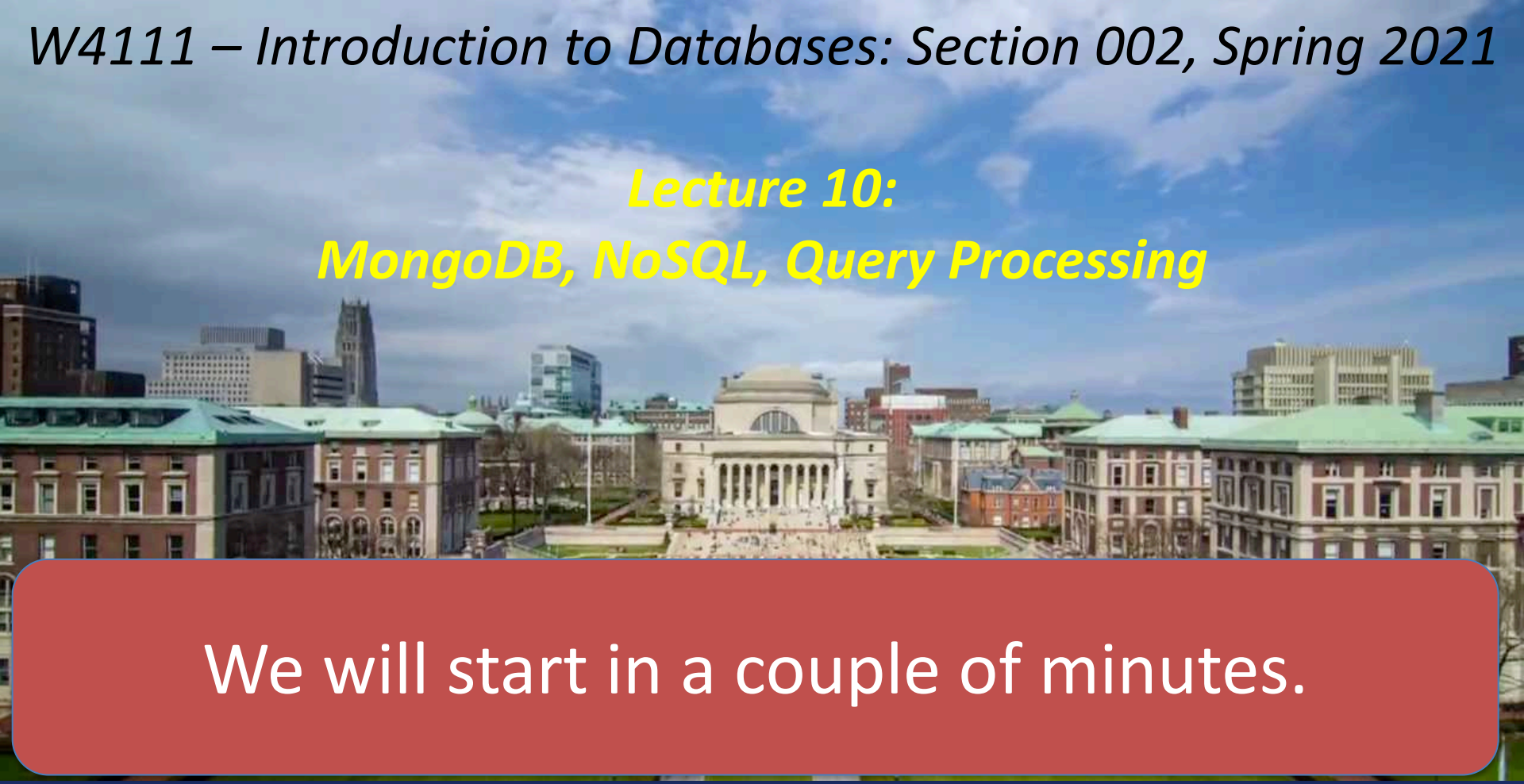
W4111 – Introduction to Databases: Section 002, Spring 2021



Lecture 10:
MongoDB, NoSQL, Query Processing

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

W4111 – Introduction to Databases: Section 002, Spring 2021



Lecture 10:
MongoDB, NoSQL, Query Processing

We will start in a couple of minutes.

Contents

Contents

- Data Models and REST – Concepts and Realization
- NoSQL Models
  - Reminder
  - Documents
- A Digression – Strong and Weak Entities
- A Document Database: MongoDB
  - MongoDB relative to RDB
  - Concepts: data types, CRUD operators
  - Aggregates and piplines
- Module II: SQL Query Processing

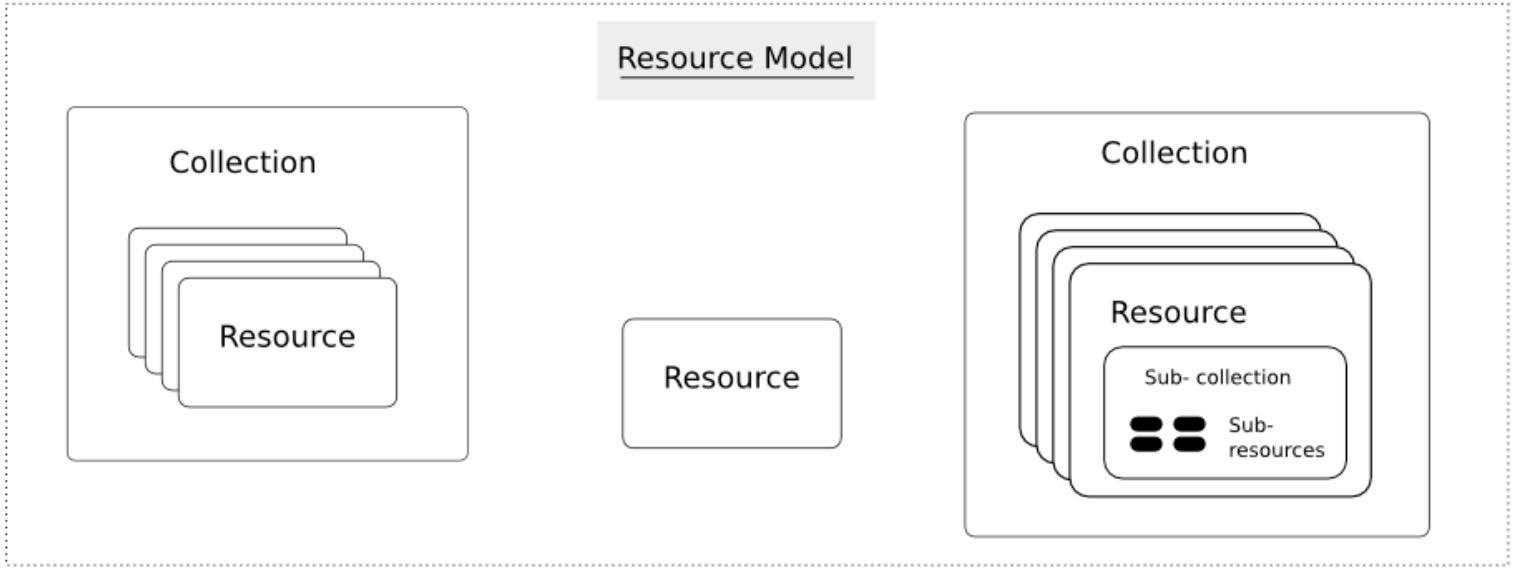
Data Models and REST

Data Modeling Concepts and REST

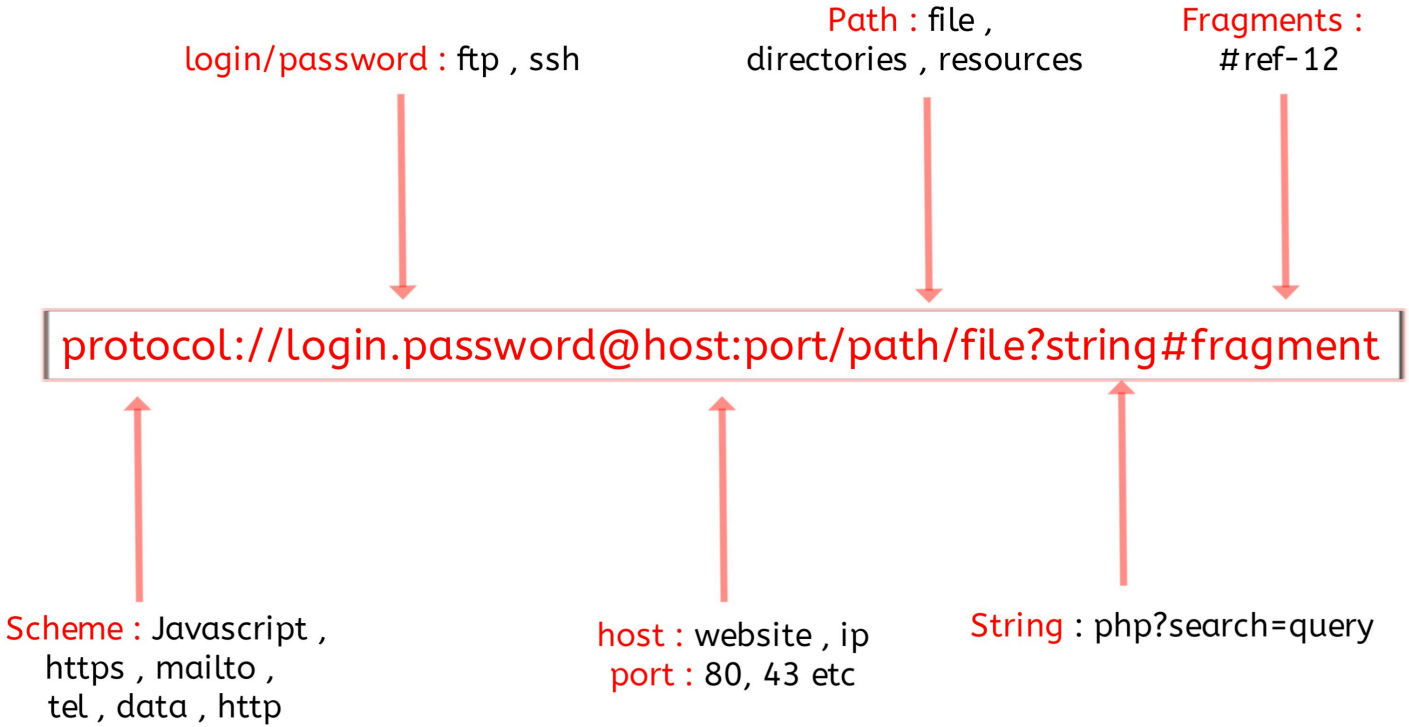
Almost any data model has the same core concepts:

- 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



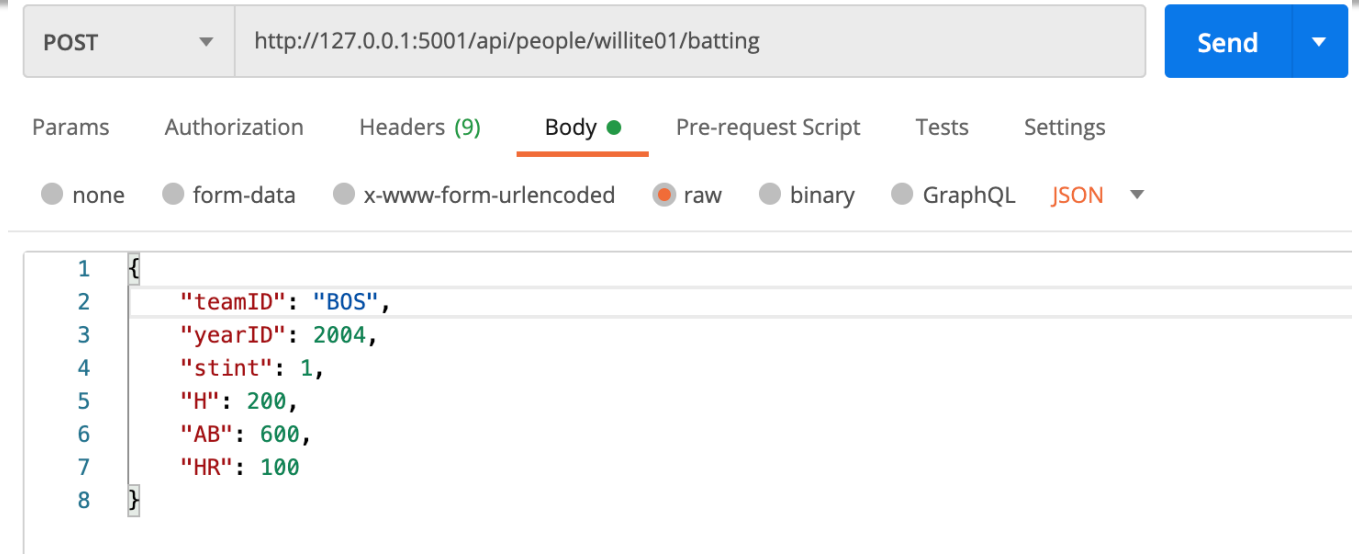
URL Mappings

- 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)

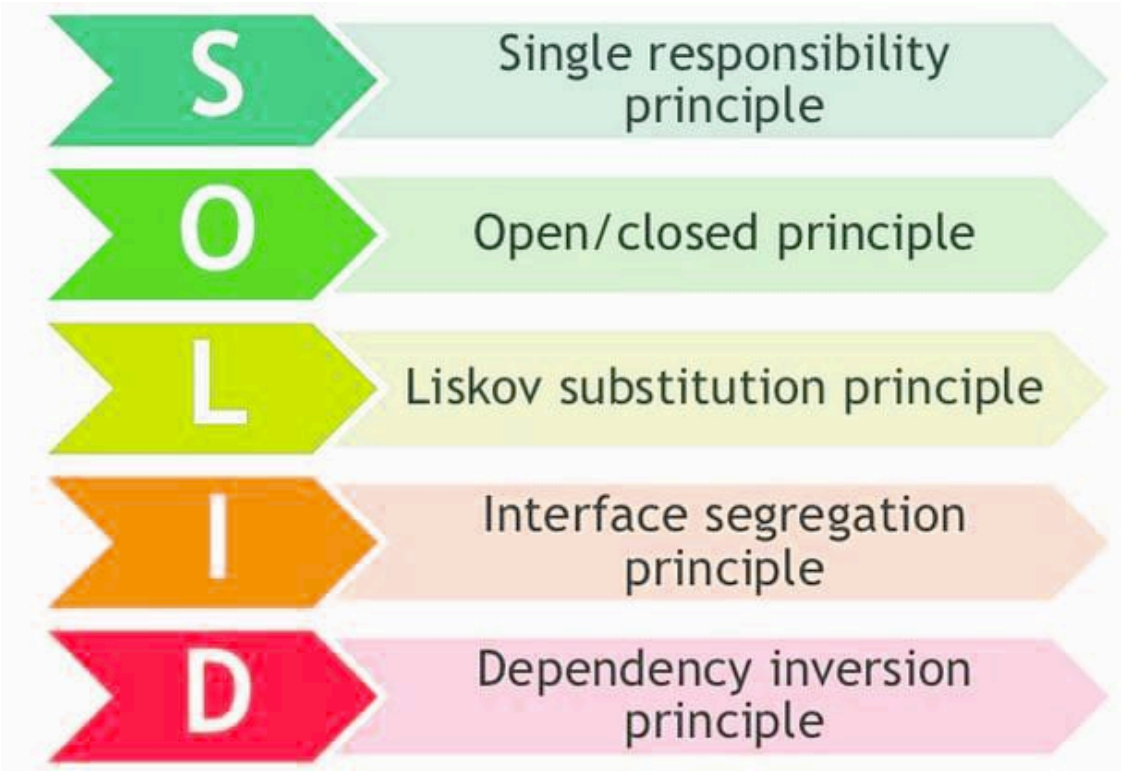
REST Method	Resource Path	Relational Operation	DB Resource
DELETE	/people	DROP TABLE	people table
POST	/people	INSERT INTO PEOPLE (...) VALUES(...)	people table people row
GET	/people/21	SHOW KEYS FROM people ...; SELECT * FROM people WHERE playerID= 21	people row
GET	/people/21/batting	SELECT batting.*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

Simplistic, Conceptual Mapping (Examples)

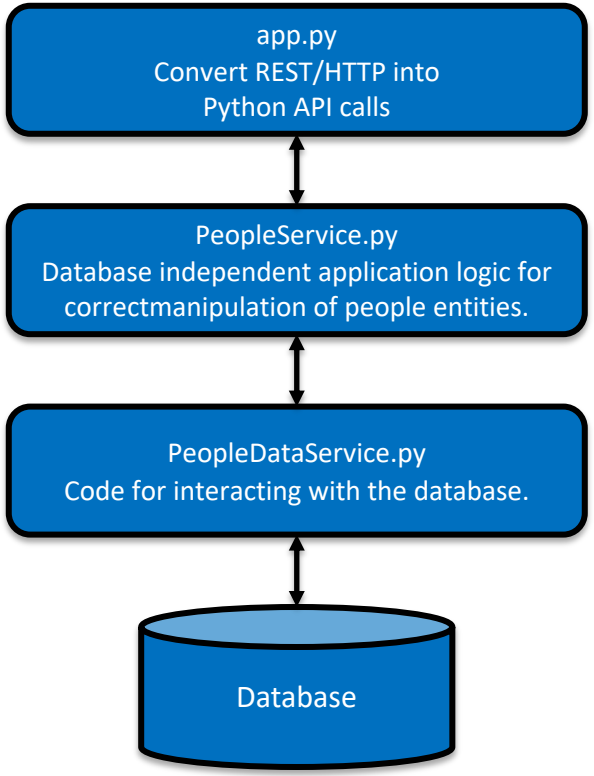



```
INSERT INTO
    batting(playerID, teamID, yearID, stint, H, AB, HR)
    VALUES ("willite01", "BOS", 2004, 1, 200, 600, 100)
```

SOLID (SW) Design Principle



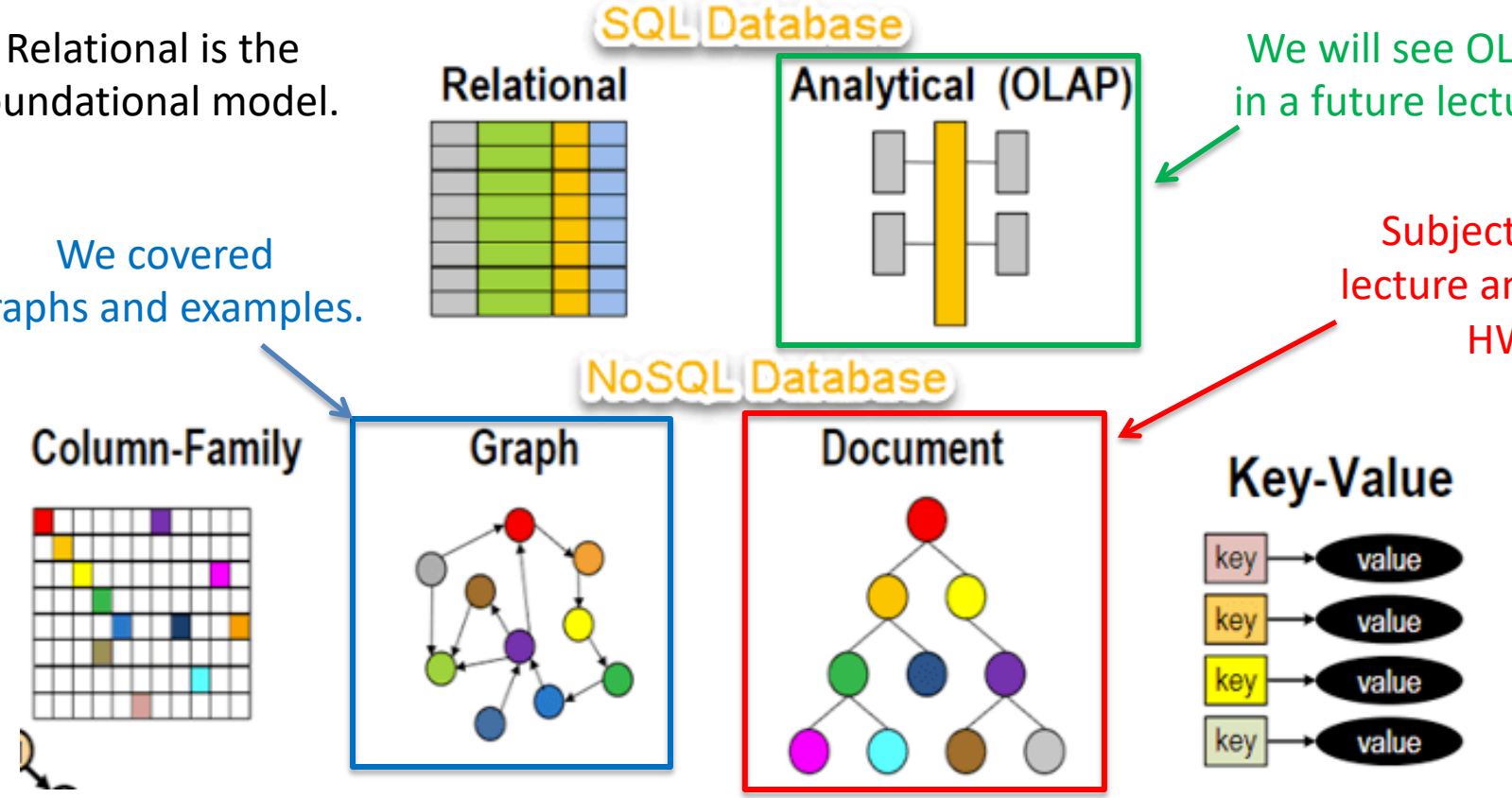
Single Responsibility



NoSQL Models

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

Relational is the foundational model.



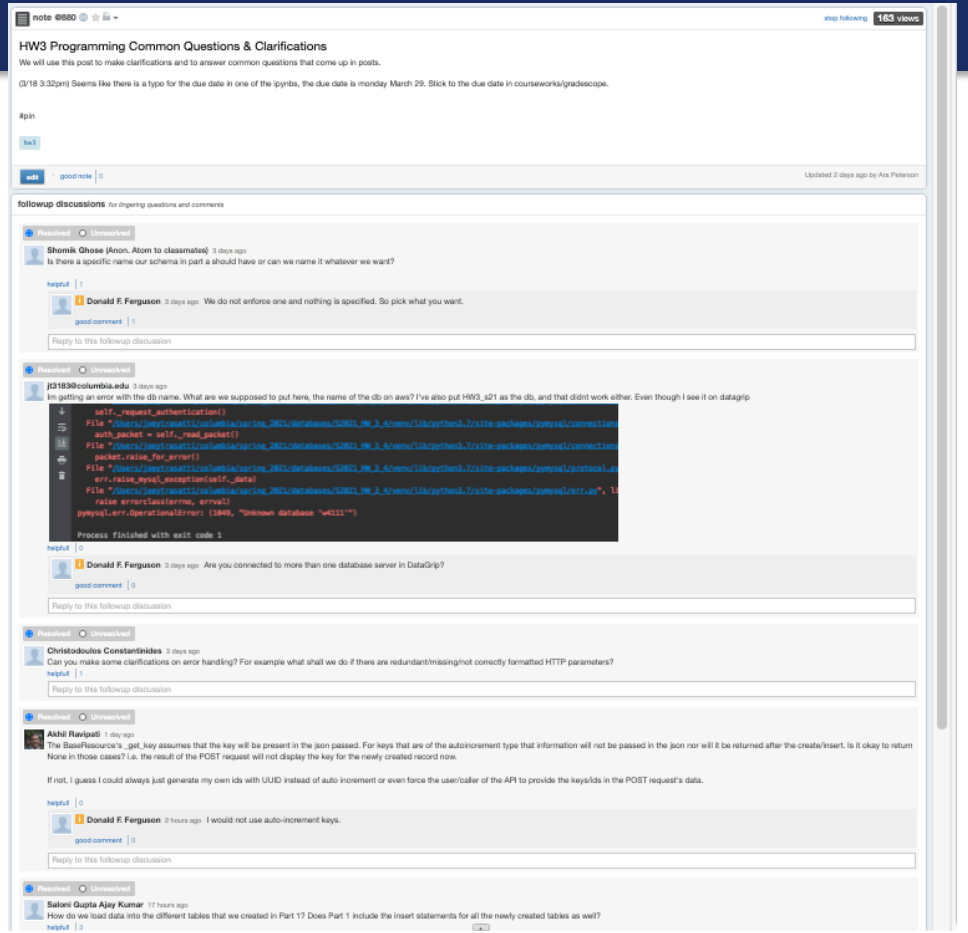
We will see OLAP in a future lecture.

We covered graphs and examples.

Subject of this lecture and part of HW4

Consider Piazza

- The data is nested:
  - Post contains multiple
    - Follow-ups which contain multiple
      - Comments
- There are multi-valued attributes, e.g. Labels: [HW1, HW2, ...]
- There are a lot of “optional values:”
  - Good Answer
  - Helpful
  - Pinned
  - Visibility
  - Poll values
  - … …
- The Relational Model does not handle this type of data well.



Documents versus Relational

Document

A document is a set of key-value pairs. Documents have dynamic schema. Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.

The following table shows the relationship of RDBMS terminology with MongoDB.

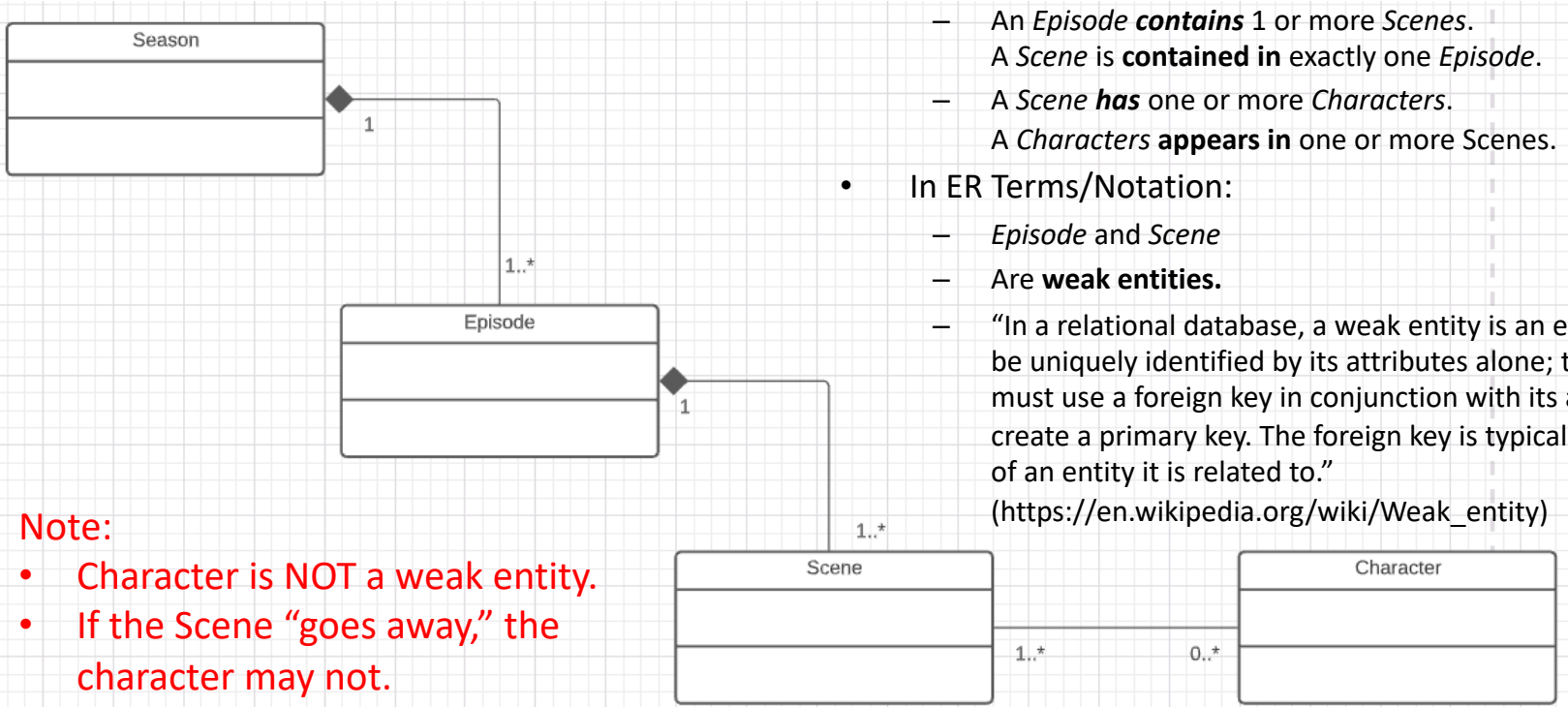
RDBMS	MongoDB
Database	Database
Table	Collection
Tuple/Row	Document
column	Field
Table Join	Embedded Documents
Primary Key	Primary Key (Default key _id provided by MongoDB itself)

Ultimately, What this Comes Down to Is ... ...

- Row versus Document:
  - Relational: All rows in a table MUST have the same columns with same types:
  - Document: Documents can have arbitrary set of attributes.
- Column versus Field:
  - Relational: A column must be an atomic type, e.g. string, number, date, ...
  - Document: A field may be:
    - An atomic type.
    - A List (i.e. array)
    - A Map (i.e. dictionary)
- Nesting:
  - Rows do not contain rows.
  - Documents can contain documents.

A Digression: Strong versus Weak Entities

The Dreaded Game of Thrones Data



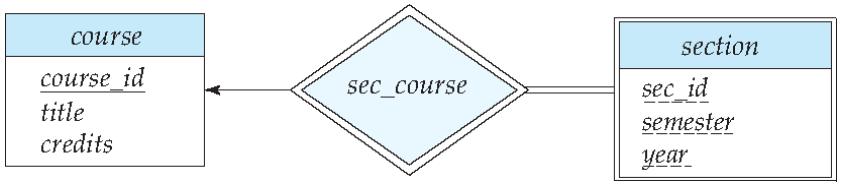
- UML Notation:
  - A *Season* ***contains*** 1 or more *Episodes*.
    An *Episode* is **contained in** exactly one *Season*.
  - An *Episode* ***contains*** 1 or more *Scenes*.
    A *Scene* is **contained in** exactly one *Episode*.
  - A *Scene* ***has*** one or more *Characters*.
    A *Characters* **appears in** one or more Scenes.
- In ER Terms/Notation:
  - *Episode* and *Scene*
  - Are **weak entities.**
  - "In a relational database, a weak entity is an entity that cannot be uniquely identified by its attributes alone; therefore, it must use a foreign key in conjunction with its attributes to create a primary key. The foreign key is typically a primary key of an entity it is related to."
    (https://en.wikipedia.org/wiki/Weak_entity)

Note:

- Character is NOT a weak entity.
- If the Scene "goes away," the character may not.

Expressing Weak Entity Sets

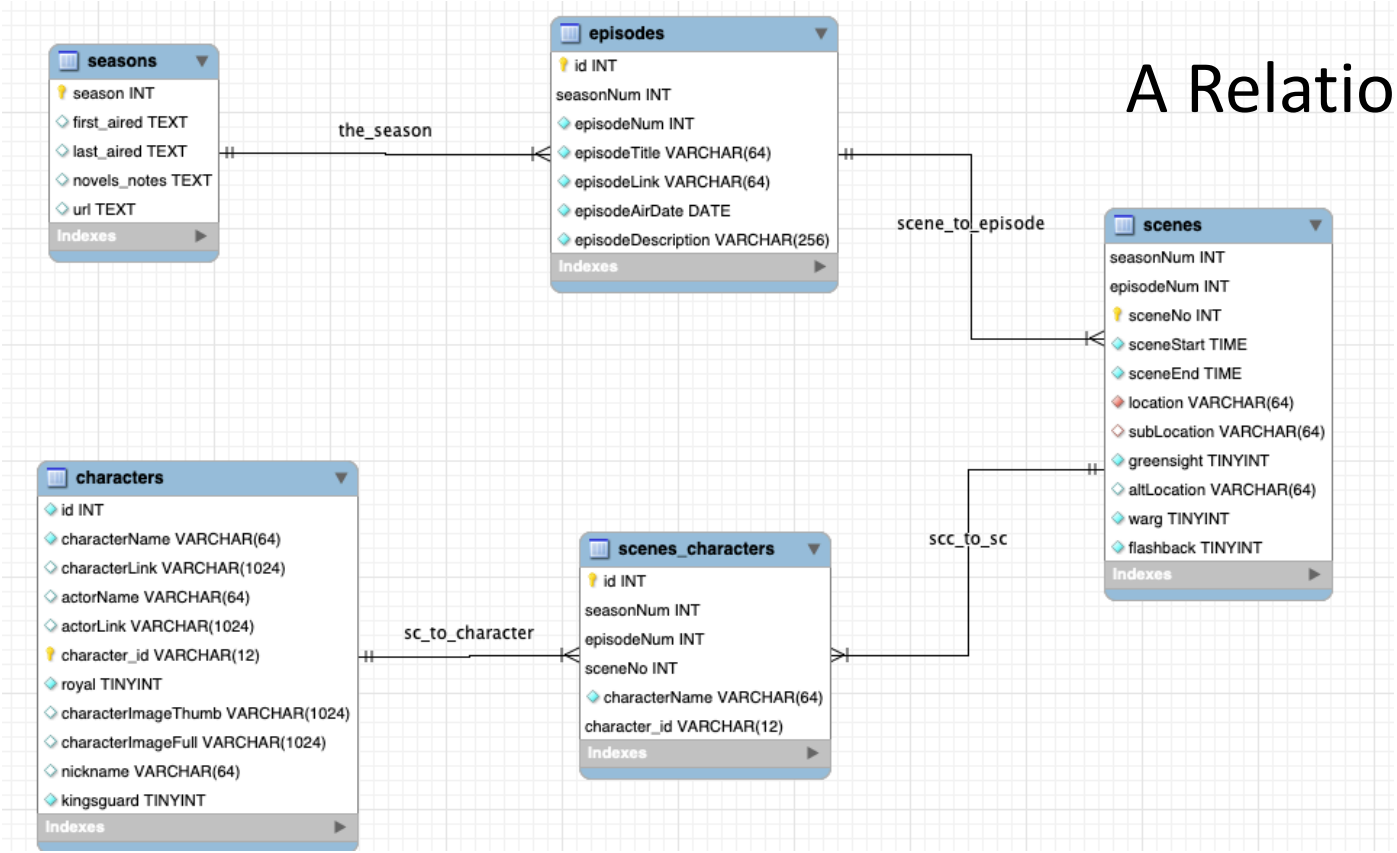
- In E-R diagrams, a weak entity set is depicted via a double rectangle.
- We underline the discriminator of a weak entity set with a dashed line.
- The relationship set connecting the weak entity set to the identifying strong entity set is depicted by a double diamond.
- Primary key for *section* – (*course_id, sec_id, semester, year*)



Notes:

- A **weak entity set** is one whose existence is dependent on another entity, called its **identifying entity.**
- There is no standard representation in Crow's Foot Notation.
- My SQL uses the concept of *identifying relationship.*

GOT Seasons, Episodes, Scenes, Characters



A Relational Model

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

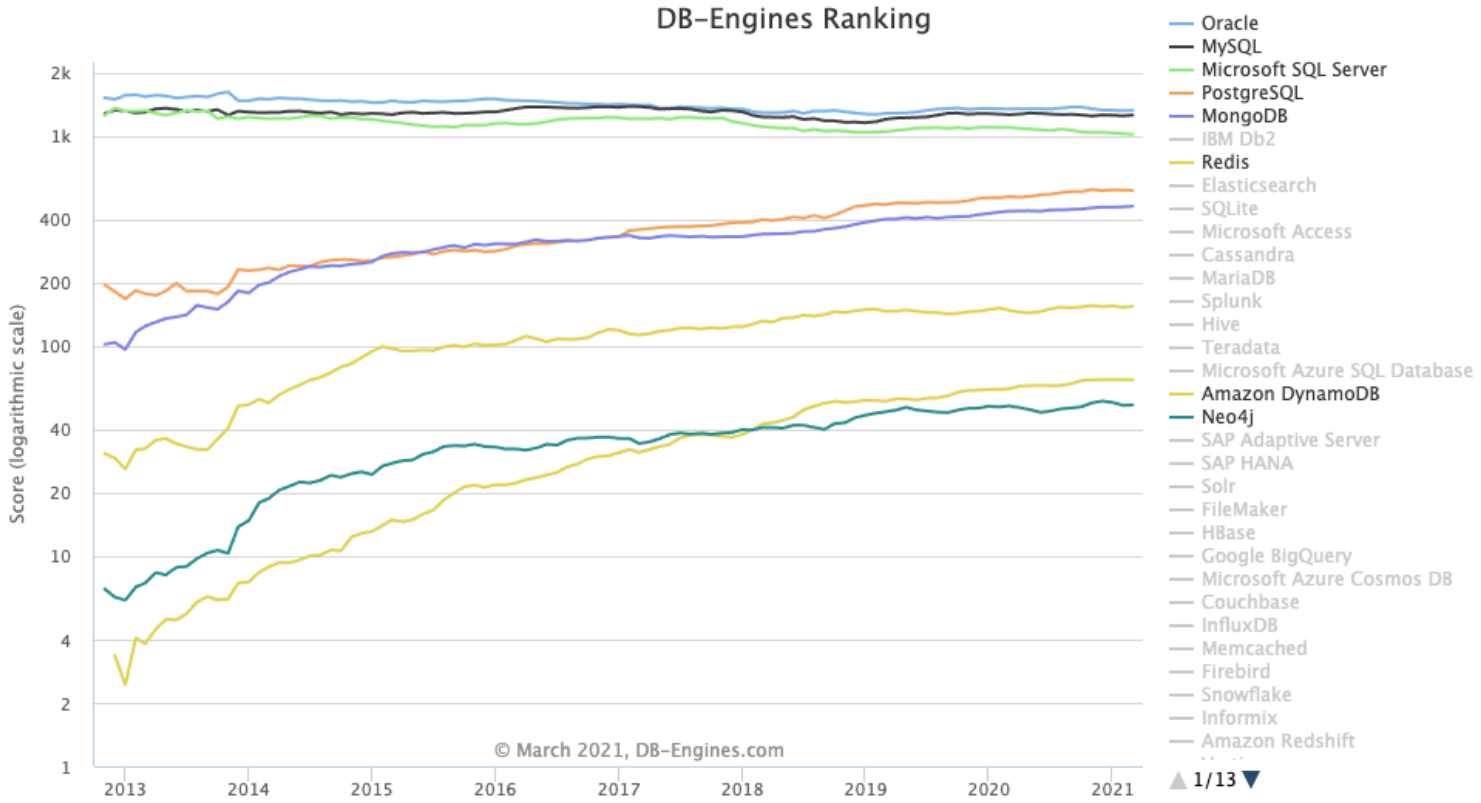
Database Adoption (https://db-engines.com/en/ranking)

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	+
5.	5.	5.	MongoDB	Document, Multi-model	462.39	+3.44	+
6.	6.	6.	IBM Db2	Relational, Multi-model	156.01	-1.60	
7.	7.	↑ 8.	Redis	Key-value, Multi-model	154.15	+1.58	
8.	8.	↓ 7.	Elasticsearch	Search engine, Multi-model	152.34	+1.34	
9.	9.	↑ 10.	SQLite	Relational	122.64	-0.53	
10.	↑ 11.	↓ 9.	Microsoft Access	Relational	118.14	+3.97	
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

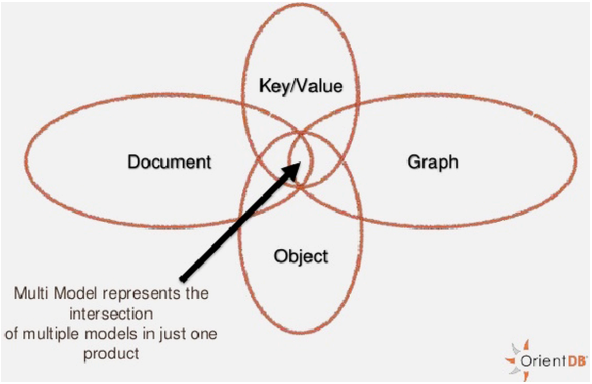
Multi-model means that the DB engine supports more than one core approach, e.g. RDB and Document

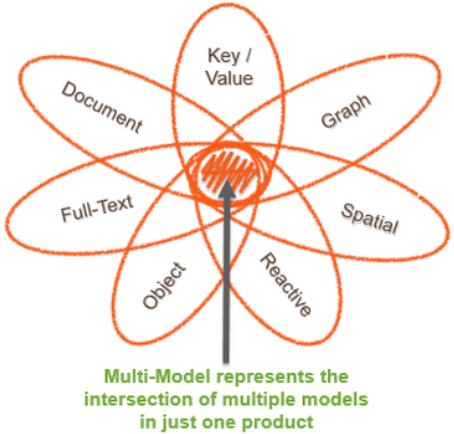
Database Adoption (https://db-engines.com/en/ranking)



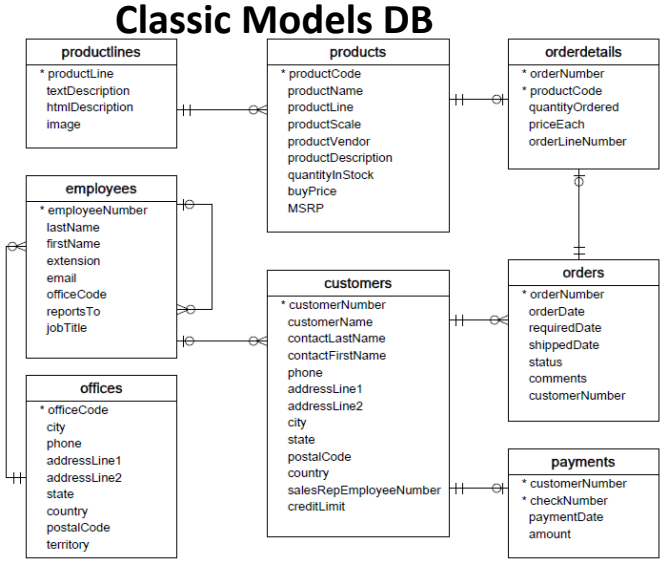
Multi-model Database

- "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, ...





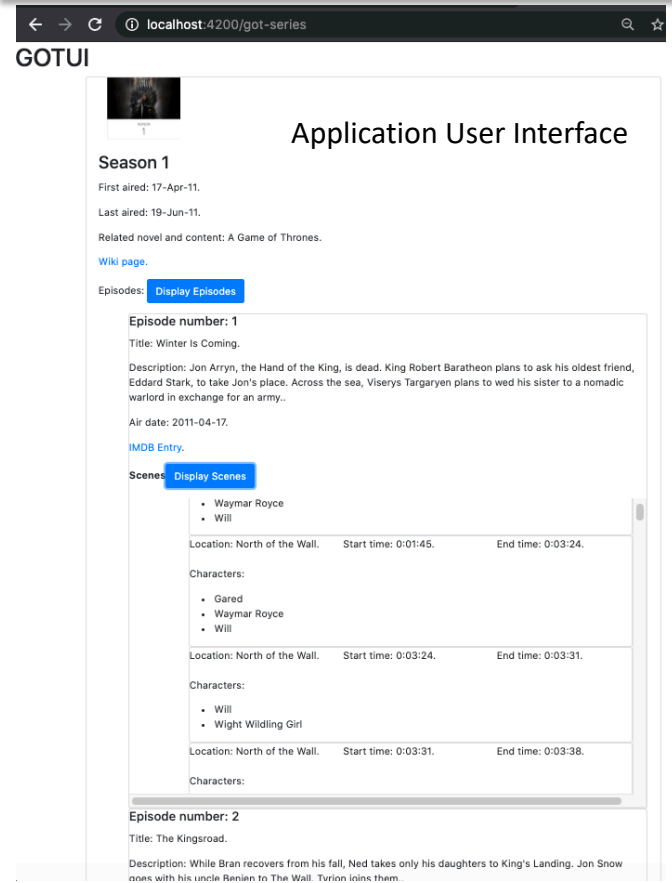
Motivation: Some Datamodels are Nested

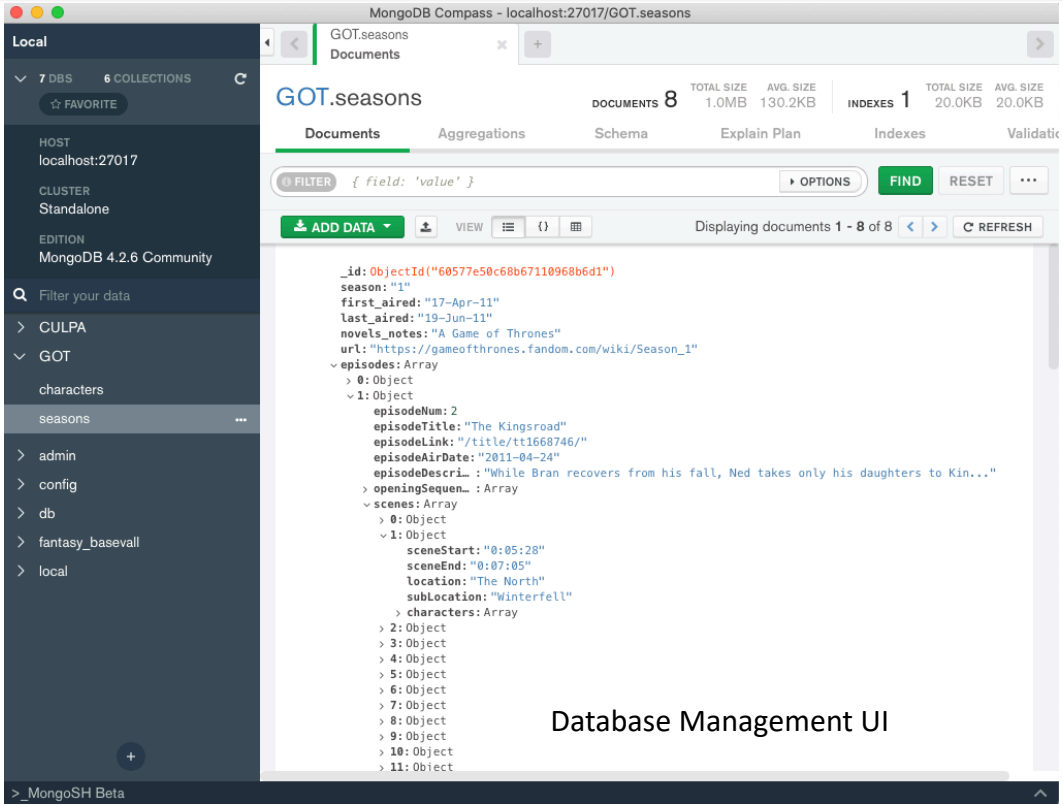


The way people think about orders.

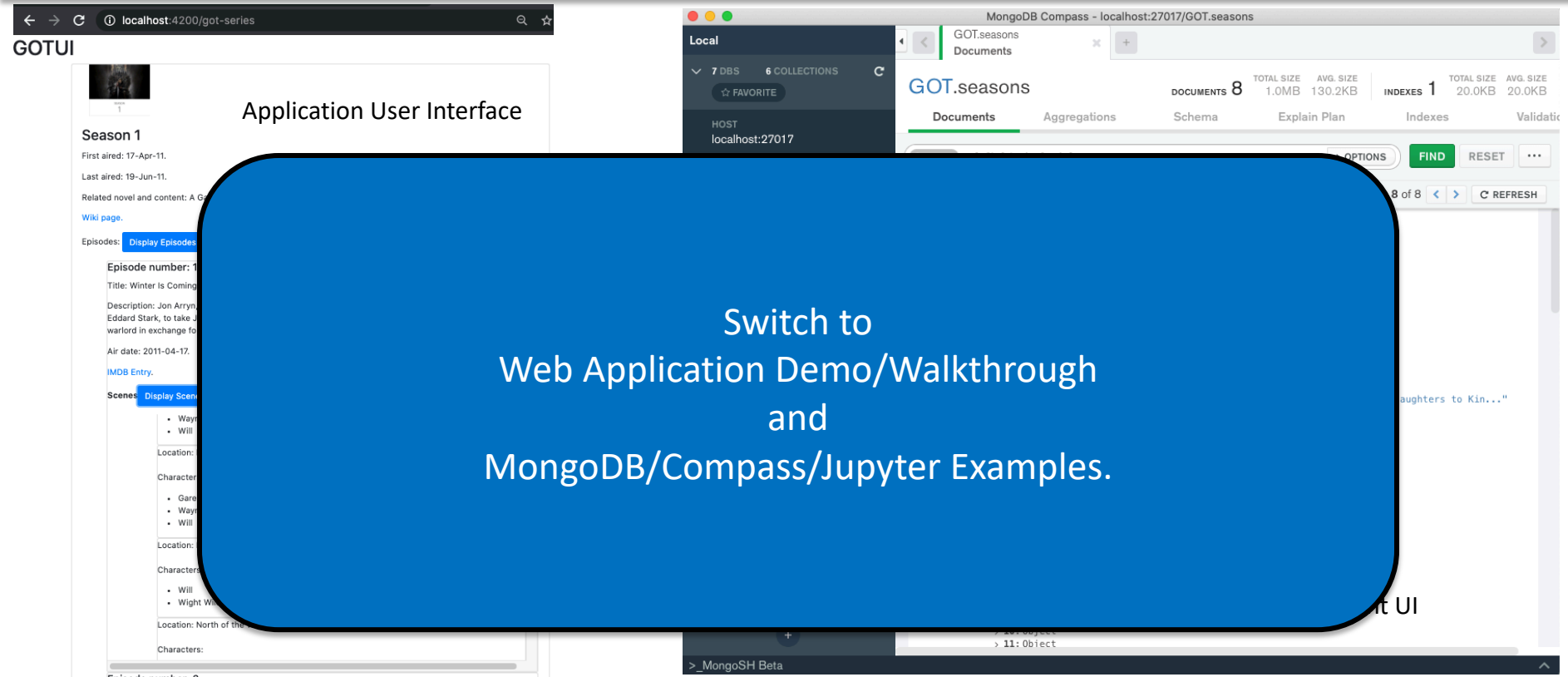
- classicmodels is a relational database (https://www.mysqltutorial.org/mysql-sample-database.aspx/)
- Relational is good for many, many things and is a great model, but some user perspectives expose data differently. In this case: table versus document (form)

Documents





Documents



Switch to
Web Application Demo/Walkthrough
and
MongoDB/Compass/Jupyter Examples.

MongoDB Overview

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

Switch to notebook.

Core Operations

Basic Operations:

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

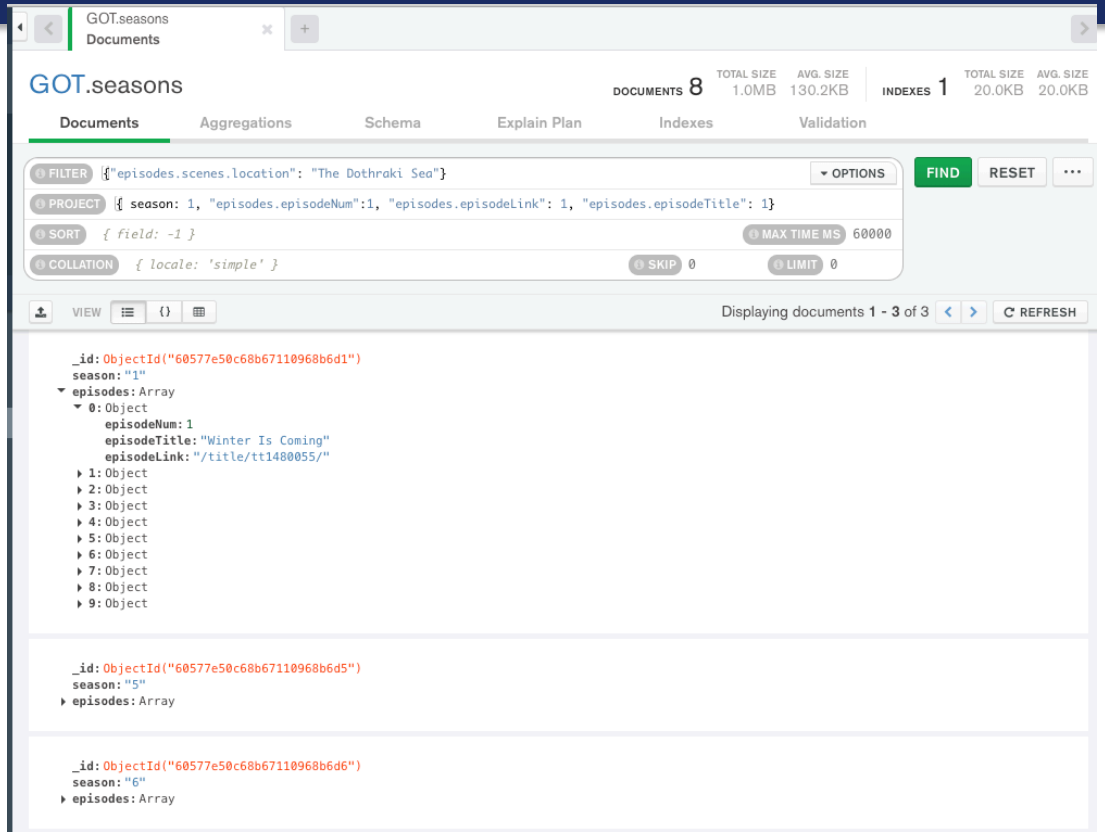
More Advanced Concepts:

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

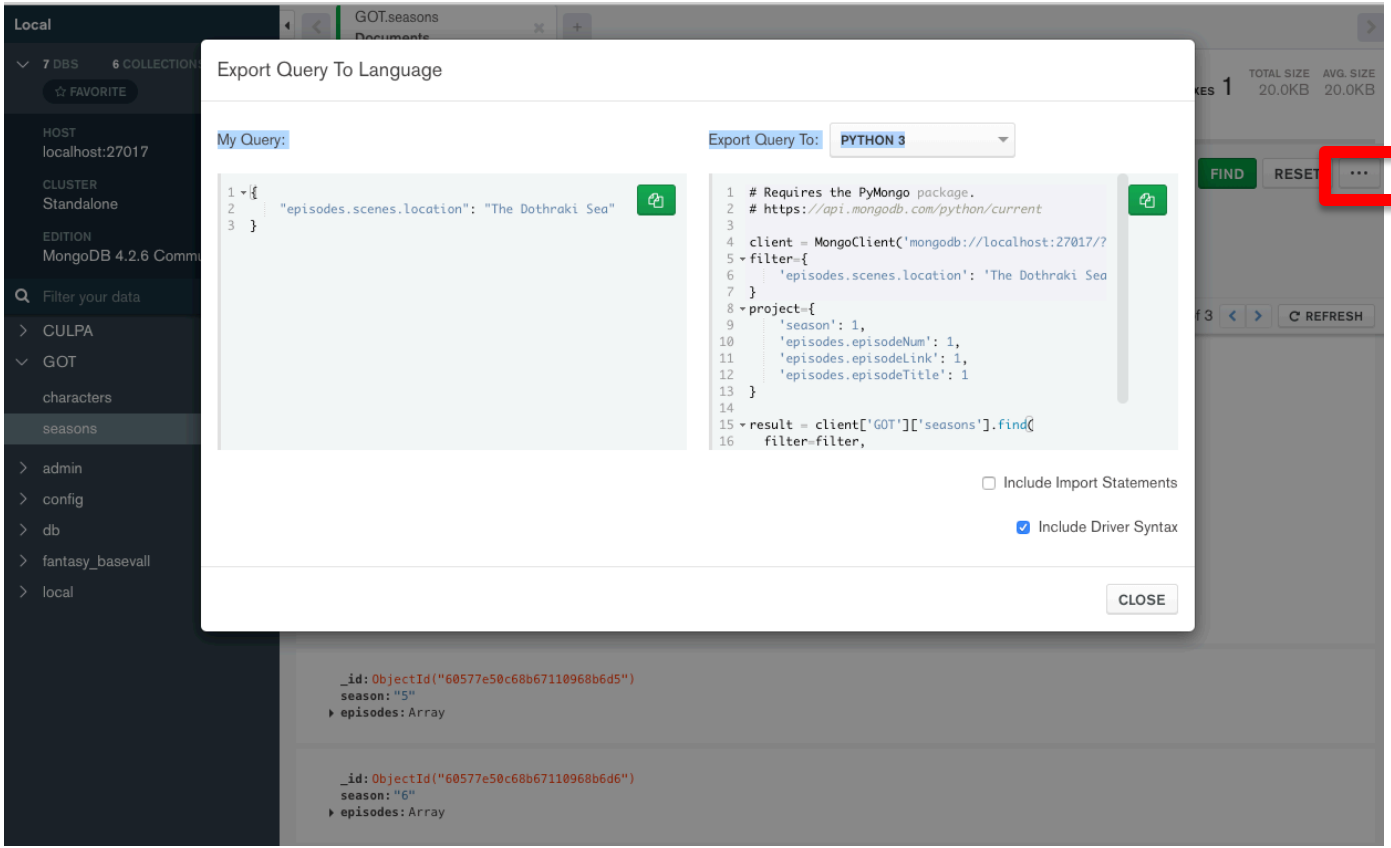
We will just cover the basics for now and may cover more things in HW or other lectures.

find()

- Note:
  - MongoDB use 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?



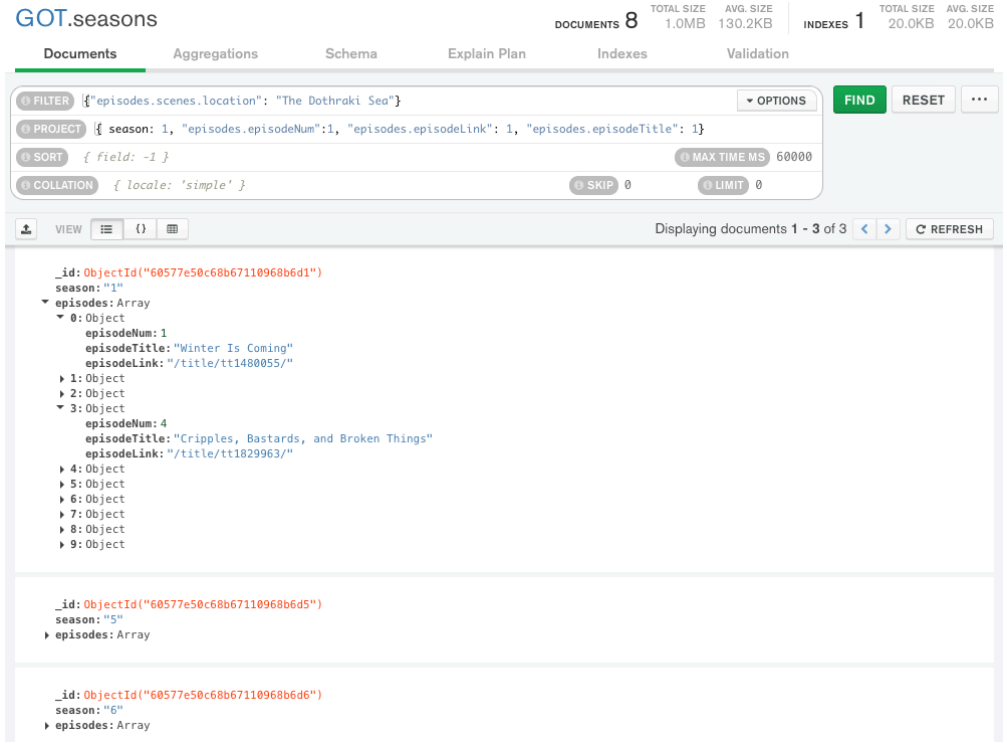
Generate Code



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

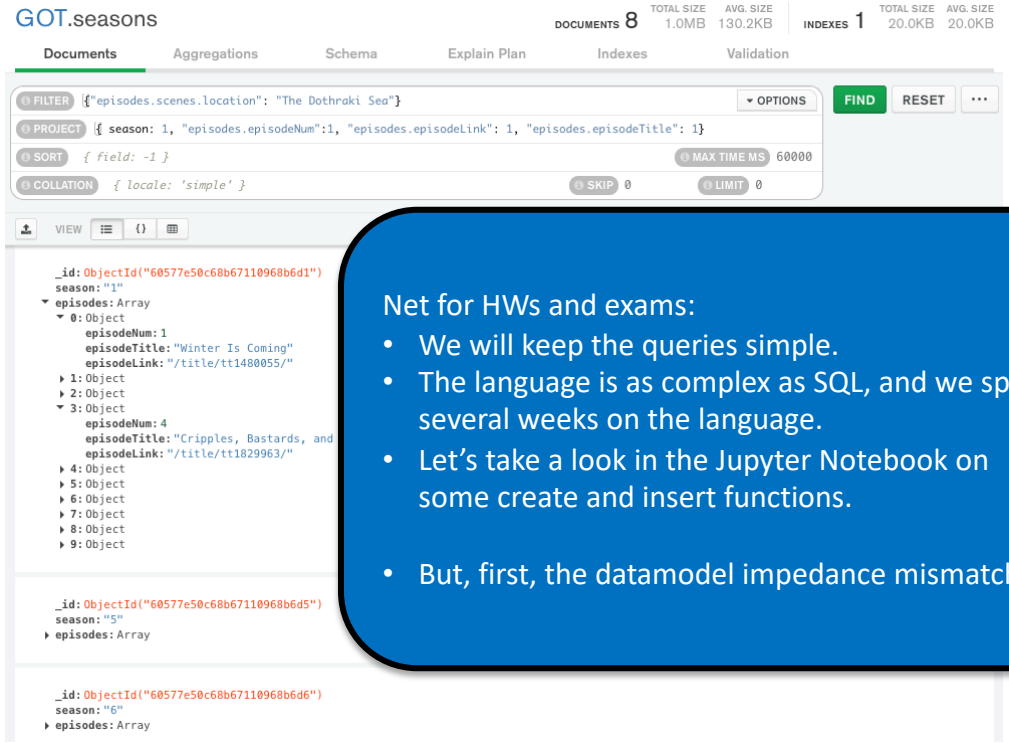
- Switch to Notebook

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 d 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.

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

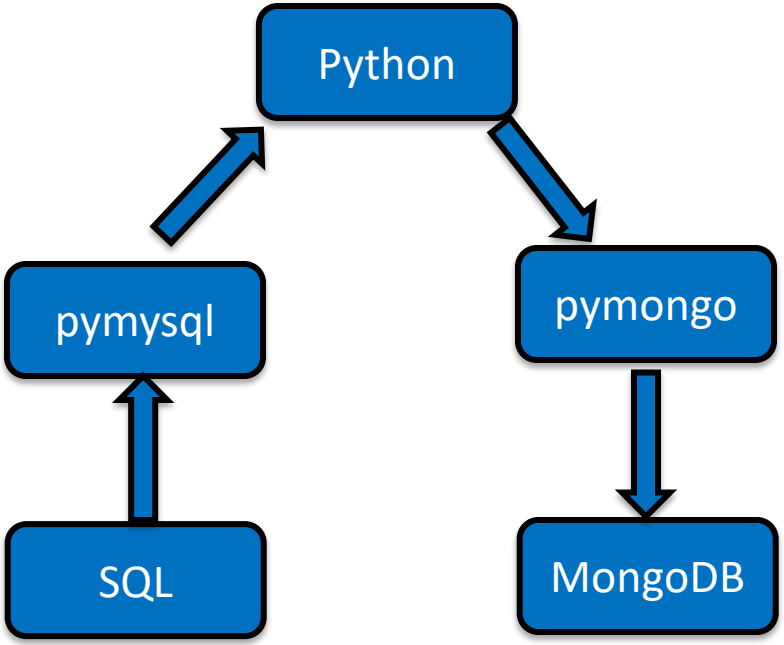
Impedance mismatch is the term used to refer to the problems that occurs due to differences between the database model and the programming language model. The practical relational model has 3 components these are:

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

More Fun – Data Types

MongoDB supports many datatypes. Some of them are −

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



(Some) MongoDB CRUD Operations

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

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

- findOne()
- find_one()

There are good online tutorials:

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

(Some) MongdoDB Pipeline Operators

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

Aggregation operators

- Pipeline and Expression operators

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

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

MongoDB Checkpoint

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

Module II: Query Processing

Query Processing Overview

Query Compilation

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

Preview of Query Compilation

To set the context for query execution, we offer a very brief outline of the content of the next chapter. Query compilation is divided into the three major steps shown in Fig. 15.2.

a) *Parsing.* A *parse tree* for the query is constructed.

b) *Query Rewrite.* The parse tree is converted to an initial query plan, which is usually an algebraic representation of the query. This initial plan is then transformed into an equivalent plan that is expected to require less time to execute.

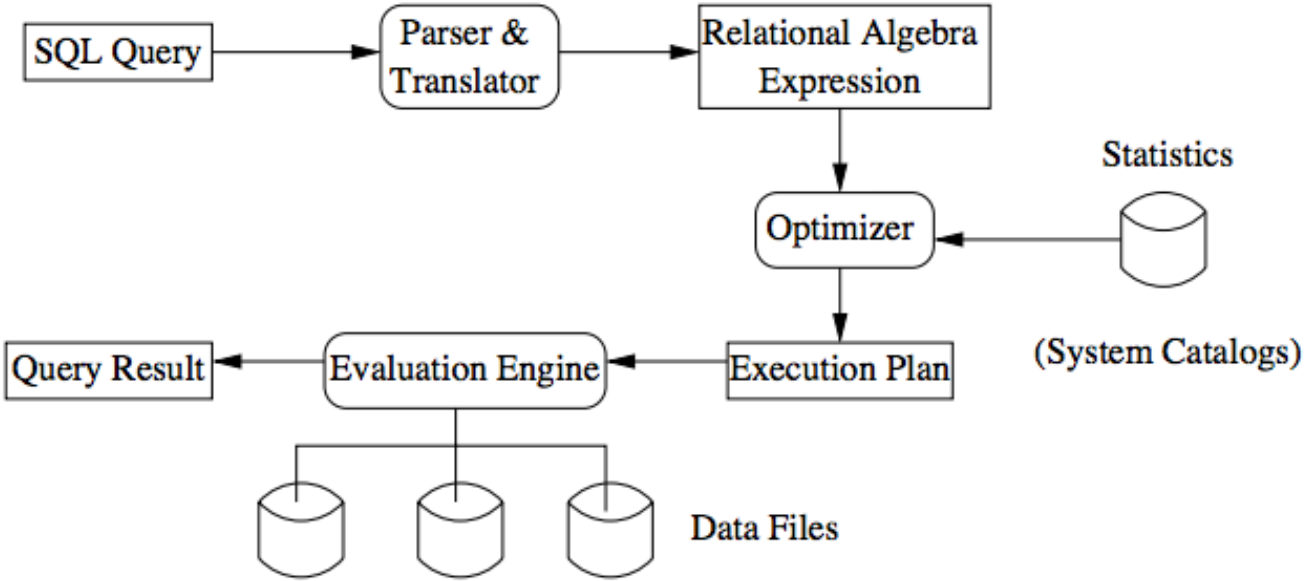
c) *Physical Plan Generation.* The abstract query plan from (b), often called a *logical query plan*, is turned into a *physical query plan* by selecting algorithms to implement each of the operators of the logical plan, and by selecting an order of execution for these operators. The physical plan, like the result of parsing and the logical plan, is represented by an expression tree. The physical plan also includes details such as how the queried relations are accessed, and when and if a relation should be sorted.

Parsing and Execution

- Parser/Translator
  - Verifies syntax correctness and generates a *parse tree.*
  - Converts to *logical plan tree* that defines how to execute the query.
    - Tree nodes are *operator(tables, parameters)*
    - Edges are the flow of data "up the tree" from node to node.
- Optimizer
  - Modifies the logical plan to define an improved execution.
  - Query rewrite/transformation.
  - Determines *how* to choose among multiple implementations of operators.
- Engine
  - Executes the plan
  - May modify the plan to *optimize* execution, e.g. using indexes.

Query Processing Overview

Basic Steps in Processing an SQL Query



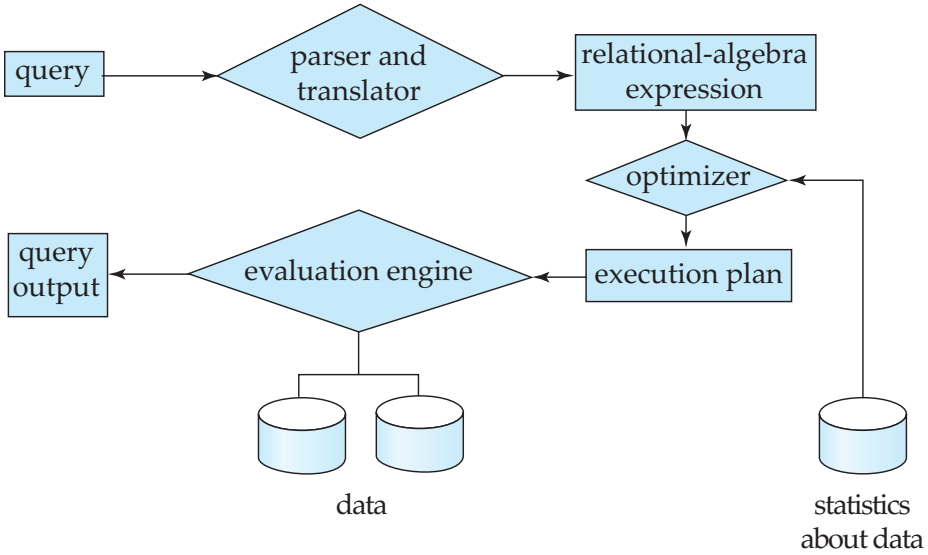
Chapter 15
From the Book

Chapter 15: Query Processing

- Overview
- Measures of Query Cost
- Selection Operation
- Sorting
- Join Operation
- Other Operations
- Evaluation of Expressions

Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



Basic Steps in Query Processing (Cont.)

- Parsing and translation
  - translate the query into its internal form. This is then translated into relational algebra.
  - Parser checks syntax, verifies relations
- Evaluation
  - The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.

Basic Steps in Query Processing: Optimization

- A relational algebra expression may have many equivalent expressions
  - E.g., $\sigma_{salary<75000}(\Pi_{salary}(instructor))$ is equivalent to $\Pi_{salary}(\sigma_{salary<75000}(instructor))$
- Each relational algebra operation can be evaluated using one of several different algorithms
  - Correspondingly, a relational-algebra expression can be evaluated in many ways.
- Annotated expression specifying detailed evaluation strategy is called an **evaluation-plan**. E.g.,:
  - Use an index on *salary* to find instructors with salary < 75000,
  - Or perform complete relation scan and discard instructors with salary ≥ 75000

Basic Steps: Optimization (Cont.)

- **Query Optimization**: Amongst all equivalent evaluation plans choose the one with lowest cost.
  - Cost is estimated using statistical information from the database catalog
    - e.g.. number of tuples in each relation, size of tuples, etc.
- In this chapter we study
  - How to measure query costs
  - Algorithms for evaluating relational algebra operations
  - How to combine algorithms for individual operations in order to evaluate a complete expression
- In Chapter 16
  - We study how to optimize queries, that is, how to find an evaluation plan with lowest estimated cost

Measures of Query Cost

- Many factors contribute to time cost
  - *disk access, CPU*, and network *communication*
- Cost can be measured based on
  - **response time**, i.e. total elapsed time for answering query, or
  - total **resource consumption**
- We use total resource consumption as cost metric
  - Response time harder to estimate, and minimizing resource consumption is a good idea in a shared database
- We ignore CPU costs for simplicity
  - Real systems do take CPU cost into account
  - Network costs must be considered for parallel systems
- We describe how estimate the cost of each operation
  - We do not include cost to writing output to disk

Measures of Query Cost

- Disk cost can be estimated as:
  - Number of seeks * average-seek-cost
  - Number of blocks read * average-block-read-cost
  - Number of blocks written * average-block-write-cost
- For simplicity we just use the **number of block transfers** *from disk and the* **number of seeks** as the cost measures
  - t_T – time to transfer one block
    - Assuming for simplicity that write cost is same as read cost
  - t_S – time for one seek
  - Cost for b block transfers plus S seeks
    $b * t_T + S * t_S$
- t_S and t_T depend on where data is stored; with 4 KB blocks:
  - High end magnetic disk: t_S = 4 msec and t_T =0.1 msec
  - SSD: t_S = 20-90 microsec and t_T = 2-10 microsec for 4KB

Measures of Query Cost (Cont.)

- Required data may be buffer resident already, avoiding disk I/O
  - But hard to take into account for cost estimation
- Several algorithms can reduce disk IO by using extra buffer space
  - Amount of real memory available to buffer depends on other concurrent queries and OS processes, known only during execution
- Worst case estimates assume that no data is initially in buffer and only the minimum amount of memory needed for the operation is available
  - But more optimistic estimates are used in practice

Join Operation

- Several different algorithms to implement joins
  - Nested-loop join
  - Block nested-loop join
  - Indexed nested-loop join
  - Merge-join
  - Hash-join
- Choice based on cost estimate
- Examples use the following information
  - Number of records of *student*: 5,000 *takes*: 10,000
  - Number of blocks of *student*: 100 *takes*: 400

- R JOIN L = L JOIN R
  - R is scan table
  - L is probe

Nested-Loop Join

- To compute the theta join $r \bowtie_\theta s$

  for each tuple t_r **in** r **do begin**
    for each tuple t_s **in** s **do begin**
      test pair (t_r, t_s) to see if they satisfy the join condition θ
      if they do, add $t_r \cdot t_s$ to the result.
    end
  end

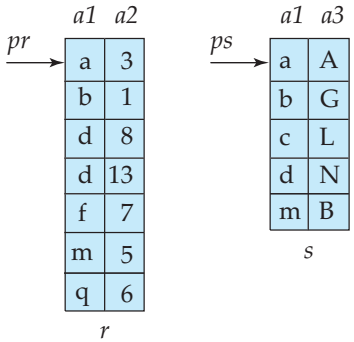
- r is called the **outer relation** and s the **inner relation** of the join.
- Requires no indices and can be used with any kind of join condition.
- Expensive since it examines every pair of tuples in the two relations.

Indexed Nested-Loop Join

- Index lookups can replace file scans if
  - join is an equi-join or natural join and
  - an index is available on the inner relation's join attribute
    - Can construct an index just to compute a join.
- For each tuple t_r in the outer relation *r,* use the index to look up tuples in *s* that satisfy the join condition with tuple t_r.
- Worst case: buffer has space for only one page of *r*, and, for each tuple in *r*, we perform an index lookup on *s.*
- Cost of the join: $b_r (t_T + t_S) + n_r * c$
  - Where *c* is the cost of traversing index and fetching all matching *s* tuples for one tuple or *r*
  - *c* can be estimated as cost of a single selection on *s* using the join condition.
- If indices are available on join attributes of both *r* and *s,* use the relation with fewer tuples as the outer relation.

Merge-Join

1. Sort both relations on their join attribute (if not already sorted on the join attributes).
2. Merge the sorted relations to join them
    1. Join step is similar to the merge stage of the sort-merge algorithm.
    2. Main difference is handling of duplicate values in join attribute — every pair with same value on join attribute must be matched
    3. Detailed algorithm in book



Merge-Join (Cont.)

- Can be used only for equi-joins and natural joins
- Each block needs to be read only once (assuming all tuples for any given value of the join attributes fit in memory
- Thus the cost of merge join is:

  $b_r + b_s$ block transfers $+ \lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil$ seeks

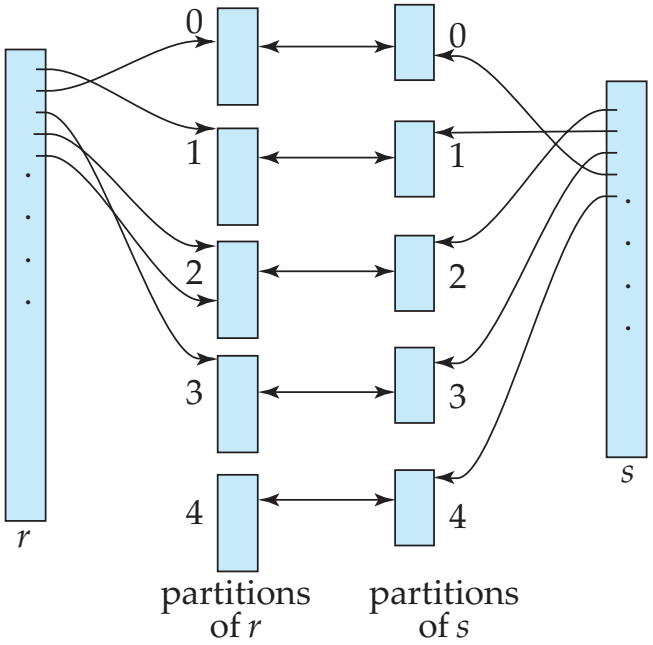
  + the cost of sorting if relations are unsorted.
- **hybrid merge-join:** If one relation is sorted, and the other has a secondary B^+-tree index on the join attribute
  - Merge the sorted relation with the leaf entries of the B^+-tree .
  - Sort the result on the addresses of the unsorted relation's tuples
  - Scan the unsorted relation in physical address order and merge with previous result, to replace addresses by the actual tuples
    - Sequential scan more efficient than random lookup

Hash-Join

- Applicable for equi-joins and natural joins.
- A hash function h is used to partition tuples of both relations
- h maps *JoinAttrs* values to {0, 1, ..., n}, where *JoinAttrs* denotes the common attributes of r and s used in the natural join.
  - $r_0, r_1, \ldots, r_n$ denote partitions of r tuples
    - Each tuple $t_r \in r$ is put in partition r_i where $i = h(t_r[JoinAttrs])$.
  - $r_0, r_1 \ldots, r_n$ denotes partitions of s tuples
    - Each tuple $t_s \in s$ is put in partition s_i, where $i = h(t_s[JoinAttrs])$.
- *Note:* In book, Figure 12.10 r_i is denoted as H_{ri}, s_i is denoted as H_{si} and n is denoted as n_h.

- *R JOIN L --> O(R*L)*
- *1. Build a hash index on L → O(L)*
- *O (R) * O(1) + O(L)*

Hash-Join (Cont.)



Hash-Join Algorithm

The hash-join of r and s is computed as follows.

1. Partition the relation s using hashing function h. When partitioning a relation, one block of memory is reserved as the output buffer for each partition.
2. Partition r similarly.
3. For each i:
   (a) Load s_i into memory and build an in-memory hash index on it using the join attribute. This hash index uses a different hash function than the earlier one h.
   (b) Read the tuples in r_i from the disk one by one. For each tuple t_r locate each matching tuple t_s in s_i using the in-memory hash index. Output the concatenation of their attributes.

Relation s is called the **build input** and r is called the **probe input**.

Hash-Join algorithm (Cont.)

- The value n and the hash function h is chosen such that each s_i should fit in memory.
  - Typically n is chosen as $\lceil b_s/M \rceil * f$ where f is a "**fudge factor**", typically around 1.2
  - The probe relation partitions s_i need not fit in memory
- **Recursive partitioning** required if number of partitions n is greater than number of pages M of memory.
  - instead of partitioning n ways, use $M - 1$ partitions for s
  - Further partition the $M - 1$ partitions using a different hash function
  - Use same partitioning method on r
  - Rarely required: e.g., with block size of 4 KB, recursive partitioning not needed for relations of < 1GB with memory size of 2MB, or relations of < 36 GB with memory of 12 MB

Other Operations

- **Duplicate elimination** can be implemented via hashing or sorting.
  - On sorting duplicates will come adjacent to each other, and all but one set of duplicates can be deleted.
  - *Optimization:* duplicates can be deleted during run generation as well as at intermediate merge steps in external sort-merge.
  - Hashing is similar – duplicates will come into the same bucket.
- **Projection**:
  - perform projection on each tuple
  - followed by duplicate elimination.

Other Operations : Aggregation

- **Aggregation** can be implemented in a manner similar to duplicate elimination.
  - **Sorting** or **hashing** can be used to bring tuples in the same group together, and then the aggregate functions can be applied on each group.
  - Optimization*:* **partial aggregation**
    - combine tuples in the same group during run generation and intermediate merges, by computing partial aggregate values
    - For count, min, max, sum: keep aggregate values on tuples found so far in the group.
      - When combining partial aggregate for count, add up the partial aggregates
    - For avg, keep sum and count, and divide sum by count at the end

Evaluation of Expressions

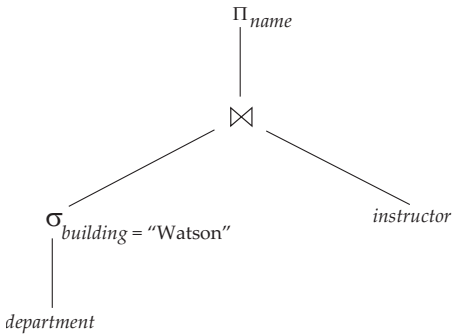
- So far: we have seen algorithms for individual operations
- Alternatives for evaluating an entire expression tree
  - **Materialization**: generate results of an expression whose inputs are relations or are already computed, **materialize** (store) it on disk. Repeat.
  - **Pipelining**: pass on tuples to parent operations even as an operation is being executed
- We study above alternatives in more detail

Materialization

- **Materialized evaluation**: evaluate one operation at a time, starting at the lowest-level. Use intermediate results materialized into temporary relations to evaluate next-level operations.
- E.g., in figure below, compute and store

$$\sigma_{building=\text{"Watson"}}(department)$$

then compute the store its join with *instructor,* and finally compute the projection on *name.*



Materialization (Cont.)

- Materialized evaluation is always applicable
- Cost of writing results to disk and reading them back can be quite high
  - Our cost formulas for operations ignore cost of writing results to disk, so
    - Overall cost = Sum of costs of individual operations + cost of writing intermediate results to disk
- **Double buffering**: use two output buffers for each operation, when one is full write it to disk while the other is getting filled
  - Allows overlap of disk writes with computation and reduces execution time

Pipelining

- **Pipelined evaluation**: evaluate several operations simultaneously, passing the results of one operation on to the next.
- E.g., in previous expression tree, don't store result of

$$\sigma_{building = \text{"Watson"}}(department)$$

  - instead, pass tuples directly to the join.. Similarly, don't store result of join, pass tuples directly to projection.
- Much cheaper than materialization: no need to store a temporary relation to disk.
- Pipelining may not always be possible – e.g., sort, hash-join.
- For pipelining to be effective, use evaluation algorithms that generate output tuples even as tuples are received for inputs to the operation.
- Pipelines can be executed in two ways: **demand driven** and **producer driven**

Pipelining (Cont.)

- In **demand driven** or **lazy** evaluation
  - system repeatedly requests next tuple from top level operation
  - Each operation requests next tuple from children operations as required, in order to output its next tuple
  - In between calls, operation has to maintain "**state**" so it knows what to return next
- In **producer-driven** or **eager** pipelining
  - Operators produce tuples eagerly and pass them up to their parents
    - Buffer maintained between operators, child puts tuples in buffer, parent removes tuples from buffer
    - if buffer is full, child waits till there is space in the buffer, and then generates more tuples
  - System schedules operations that have space in output buffer and can process more input tuples
- Alternative name: **pull** and **push** models of pipelining

Pipelining (Cont.)

- Implementation of demand-driven pipelining
  - Each operation is implemented as an **iterator** implementing the following operations
    - **open()**
      - E.g., file scan: initialize file scan
        - state: pointer to beginning of file
      - E.g., merge join: sort relations;
        - state: pointers to beginning of sorted relations
    - **next()**
      - E.g., for file scan: Output next tuple, and advance and store file pointer
      - E.g., for merge join: continue with merge from earlier state till next output tuple is found. Save pointers as iterator state.
    - **close()**