NoSQL and MongoDB

Sean Sall

June 8th, 2016

Objectives

- Compare and contrast SQL and noSQL
- Complete basic operations with Mongo

Agenda

- SQL versus NoSQL
- Mongo
  - High level overview
  - Connecting to Mongo
  - Mongo operations (and their SQL equivalents)

Why does this matter?

- NoSQL and NoSQL-like databases are popular for unstructured data
- NoSQL and NoSQL-liked databases are common to use when web-scraping

SQL and NoSQL

SQL review

- SQL allows us to interact with Relational Database Management Systems (RDBMS), which provide the ability to:
  - model relations in the data (via an Object-Relational-Model, or ORM)
    - ★ stores data about one object across multiple tables
  - query data and their relations efficiently
  - maintain data consistency and integrity
  - share data easily between programming languages

RDBMS Data Model

- A RDBMS is composed of a number of user-defined **tables**, each with **columns (fields)** and **rows (records)**
  - each column is of a certain **data type** (integer, string, date)
  - each row is an entry in the table (an observation) that holds values for each one of the columns
  - tables are specified by a **schema** that defines the structure of the data
  - we specify the table structure **ahead** of time

RDBMS Schema Visual

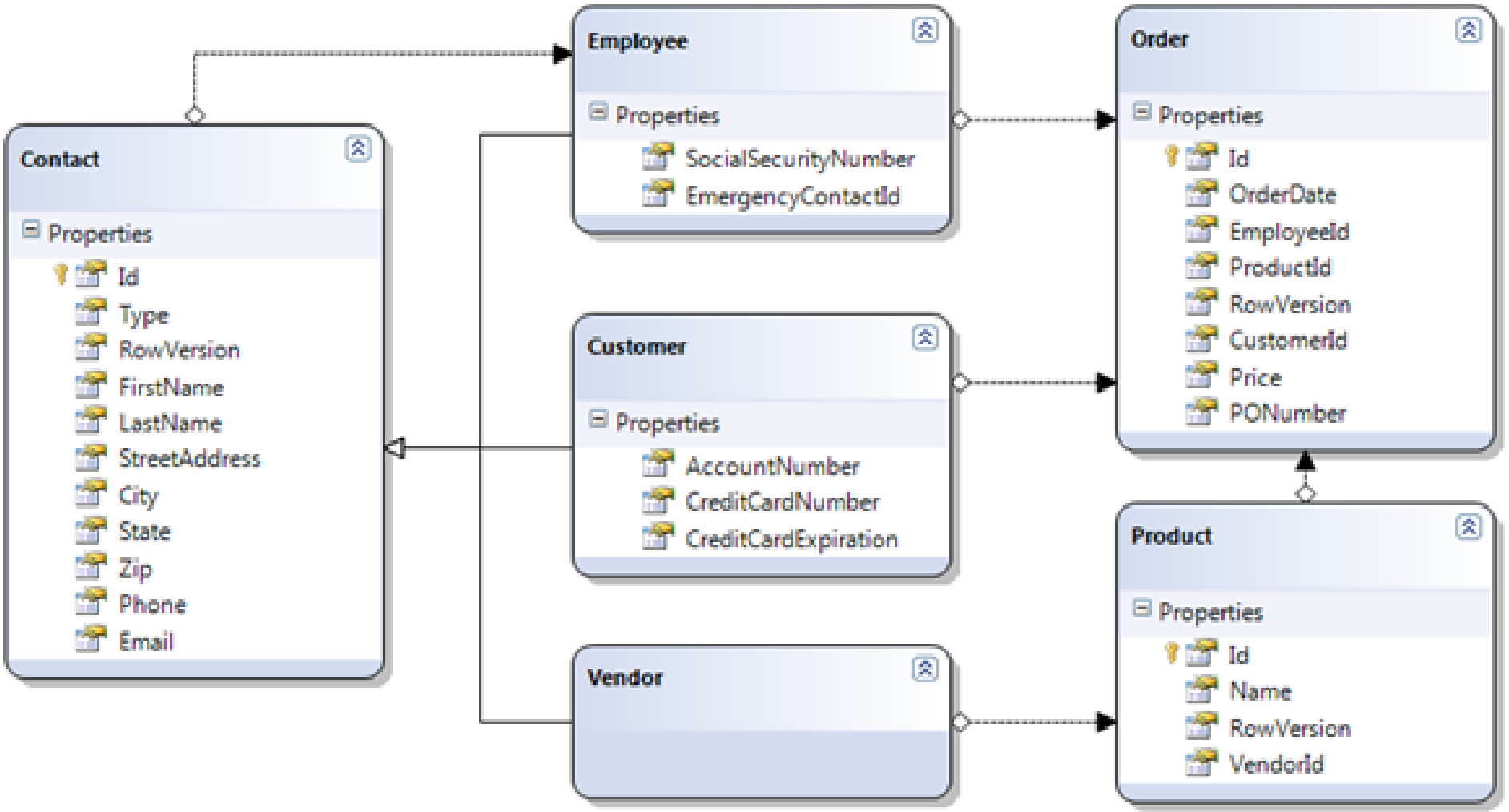


Figure 1:A Schema

NoSQL intro

- NoSQL stands for *not only SQL*
- A large number of NoSQL databases are **document-oriented**
  - each object (e.g row or **document**) in the database is stored in one place
  - each object (e.g row or **document**) can be completely different from all others
    - ★ there is no schema
- Given a lack of schema, a NoSQL database might be preferable to SQL when we have unstructured data (as we often do when pulling from the web)

MongoDB intro

- MongoDB is a flavor of NoSQL (just like PostgreSQL is a flavor of SQL)
- MongoDB is document-oriented:

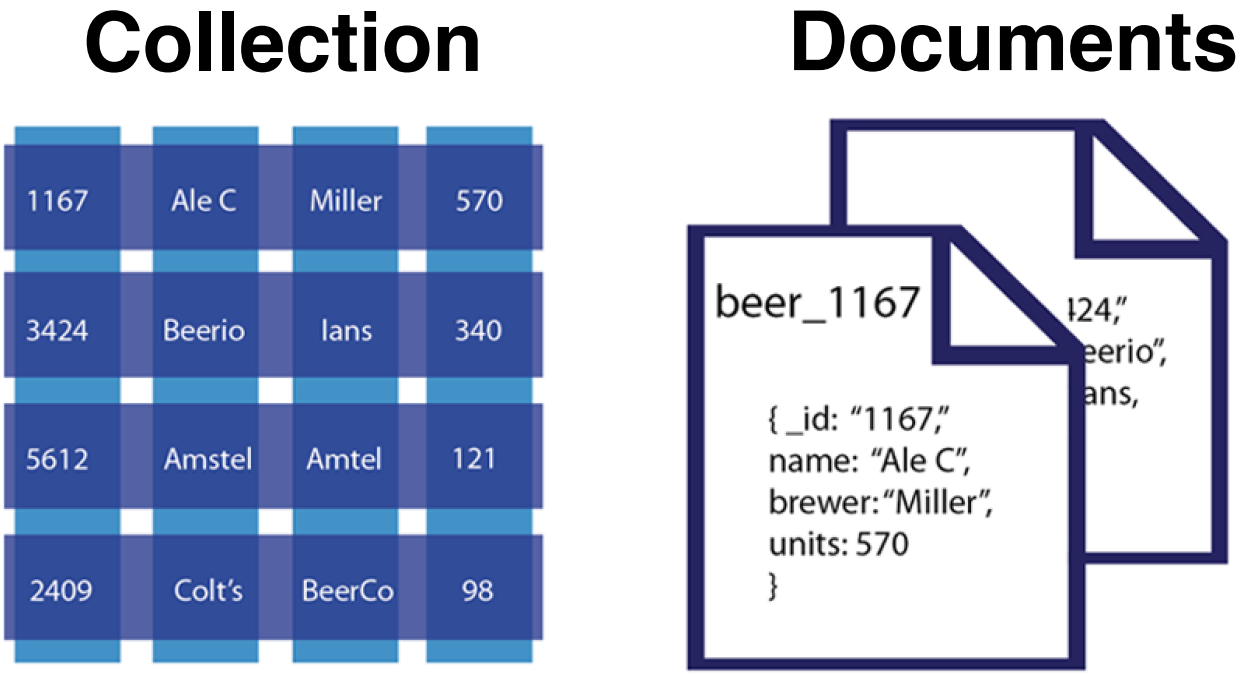


Figure 2:Document-oriented database

MongoDB terminology

- In terms of terminology, for MongoDB we have the following for our SQL equivalents:

SQL	MongoDB
database	database
table	collection
row	document
column	field

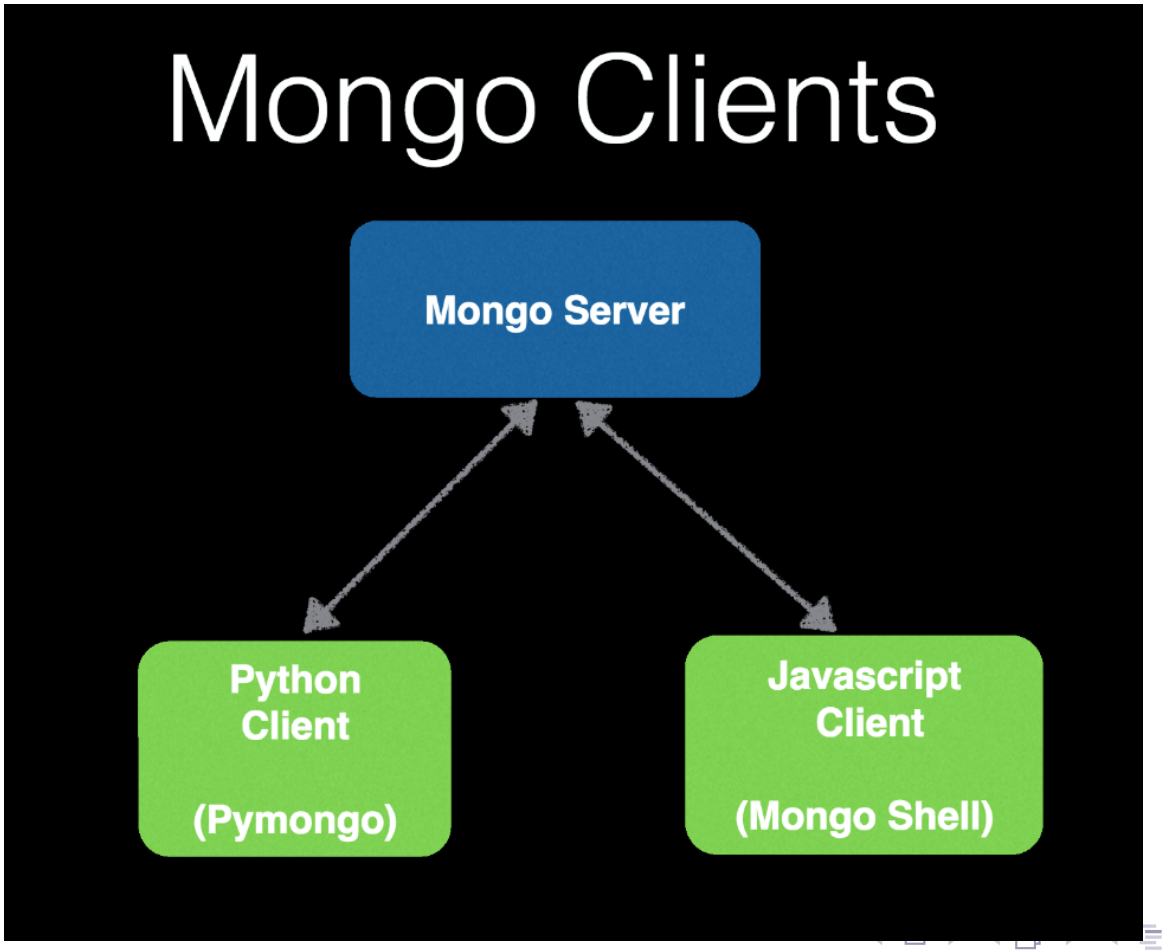
SQL versus Mongo

- With SQL, we want to prevent reduncancy in data by having tables with unique information and relations between them (normalized data).
  - creates a framework for querying with joins.
  - makes it easier to update database. Only ever have to change information in a single place.
  - can result in "simple" queries being slower, but more complex queries are often faster.
- MongoDB is a document based storage system that does not enforce normalized data. It allows data redundancies in documents (denormalized data).
  - no joins
  - changes to databases generally result in needing to change many documents
  - because of the redundancy in the documents, simple queries are generally faster, but complex queries are often slower

Mongo in Practice

Connecting to MongoDB I

- In practice, there are two main ways that we connect to Mongo:
  1. from the console using the `mongo shell`
  2. from Python using `pymongo`



Connecting to MongoDB II

Before we connect to mongo (either through Python or the `mongo shell`), we have to start a mongo server. We do that by typing the following in the terminal:

```
mongod
```

Note: The Mongo Daemon will need to occupy the terminal that you started it in for the life of the server session (so, you should run the above command in a separate terminal tab or use something like `tmux`).

Connecting to MongoDB Using the Mongo Shell

- After starting the mongo server, we can connect to a Mongo Shell in one of two ways:

```
mongo # Connects to a default database
      # (usually your username)
mongo <database_name> # Connects to the database named
                      # <database_name>
```

Mongo Commands - The basics

- Basic Mongo commands:

command	purpose
help	List top level mongo commands
db.help()	List database level mongo commands
db.collection_name.help()	List collection level mongo commands
show dbs	Obtain a list of databases
use db_name	Change the current database to db_name
show collections	Obtain list of collections in current database

Mongo Commands - Creating a database and collection

- To create a database and collection, all we have to do is switch into the database (using `use`), and then insert a record into it.

```
use class_db # Creates class_db if it doesn't exist, and
             # otherwise logs into it.
db.disney_chars.insert{name: 'Jasmine', age: 22}
```

Note: Mongo will create any databases that don't exist, as well as any collections when you try to insert into them.

Mongo Commands - The Mongo "SELECT" and "FROM"

- Using `.find` is the Mongo equivalent of SQL's `SELECT` (kind of - below we are selecting all **fields**)

```
db.disney_chars.find() # Returns all documents.
db.disney_chars.find().limit(5) # Returns first 5 documents.
```

- Note that the collection comes after the `db`, and this is the equivalent of using SQL's `FROM` and putting a table name after it

Mongo Commands - The Mongo "WHERE"

- We can specify many ways of finding observations in Mongo:

```
db.disney_chars.find({name: 'Mulan'}) # Find by single field
db.disney_chars.find({age: 26}) # Find by single field
db.disney_chars.find({age: {$exists : true }}) # Find by
                                               # presence
                                               # of field
db.disney_chars.find({friends: 'BoBo'}) # Find by whether
                                        # value in array
```

Mongo Commands - The Mongo "SELECT" II

- To specify only certain fields to keep, we have to pass those in as a second argument:

```
# Select all documents with name 'Mulan', and only return
# back the name field (_id is returned by default, so
# it'll be returned as well)
db.disney_chars.find({name: 'Mulan'}, {name: true})

# Select all documents, returning their friends field
# (_id is returned by default, so it'll be returned
# as well)
db.disney_chars.find({}, {friends: true})
```

Mongo Commands - The Mongo "SELECT" III

```
# Select all documents, returning **only** their friends
# field, and **not** _id with it.
db.disney_chars.find({}, {friends: true, _id: false})

# Select all documents, returning **only** their friends
# field, and **not** _id with it, but **only** if they
# have a friends field.
db.disney_chars.find({friends: {$exists: true}},
                     {friends: true, _id: false})
```

Mongo Commands - The Mongo "SELECT" IV

```
# Find those documents without name 'Mulan'
db.disney_chars.find({name : {$ne: 'Mulan'}})
```

- We have other operators for the other equality-like comparisons:

Operator Syntax	Meaning
$eq	Equals
$gt	Greater than
$gte	Greater than or equal to
$lt	Less than
$lte	Less than or equal to
$ne	Not equal to
$in	In (for arrays)
$nin	Not in (for arrays)

Mongo Aggregations I

- Aggregations in Mongo are a little more involved (and much less pretty) than in SQL:

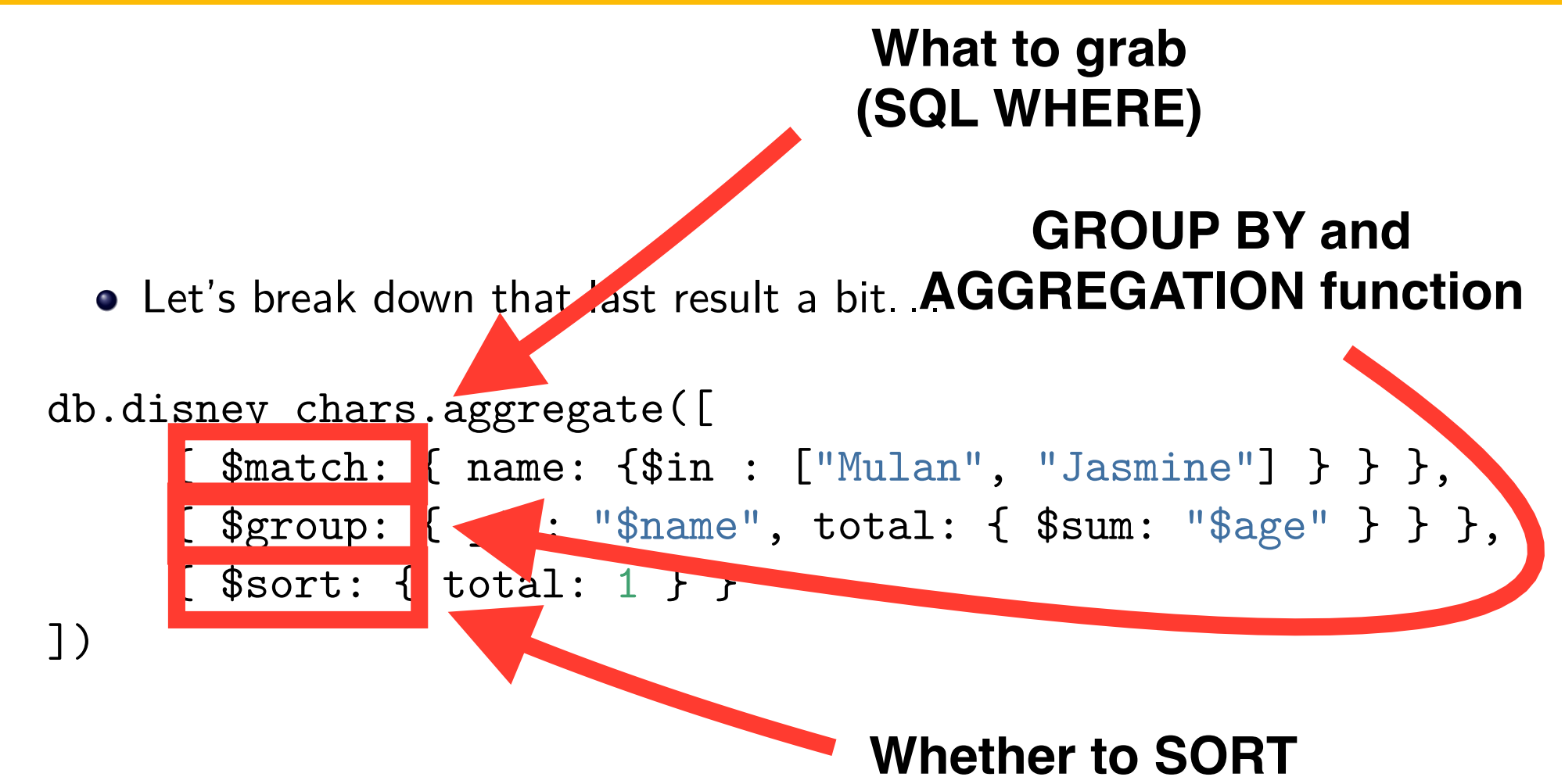
```
db.disney_chars.aggregate( [ { $group :
    {
        _id: "$name",
        total: { $sum: "$age"}
    }
}])
```

```
db.disney_chars.aggregate([
     { $match: { name: {$in : ["Mulan", "Jasmine"] } } },
     { $group: { _id: "$name", total: { $sum: "$age" } } },
     { $sort: { total: 1 } }
])
```

Mongo Aggregations I Breakdown

- Let's break down that last result a bit. . .

```
db.disney_chars.aggregate([
    { $match: { name: {$in : ["Mulan", "Jasmine"] } } },
    { $group: { _id: "$name", total: { $sum: "$age" } } },
    { $sort: { total: 1 } }
])
```



Mongo Aggregations II

- Other aggregation functions we have:

Aggregation command	Purpose
aggregate	Performs aggregation tasks
count	Counts the number of items meeting some criteria
distinct	Displays distinct values for a specified field(s)
group	Groups observations in some way

Mongo Updates

- To update a record, we can do the following:

```
# Update the **first instance** with name Mulan,
# setting their age to 29.
db.disney_chars.update({name: "Mulan"}, {$set : {age: 29}})

# Update all instances.
db.disney_chars.update({name: "Mulan"}, {$set : {age :29}},
                       {multi: true})

# Update all instances found, or insert a document
# if not found.
db.disney_chars.update({name: "Mulan"}, {$set : {age :29}},
                       {multi: true, upsert: true})
```

Mongo Updates Breakdown

What to update (like WHERE)

How to update

- Let's break down that last result a bit...

```
# Update all instances found or insert a document
# if not found.
db.disney_chars.update({name: "Mulan"}, {$set : {age :29}},
                       {multi: true, upsert: true})
```

Update all documents that match (as opposed to just the first one)

Insert a document like this if none exist

Mongo Shell allows valid JavaScript

- The Mongo Shell is technically a JavaScript shell, meaning it allows any valid JavaScript (JS) code. That includes loops:

```
#  Update all those with name "Mulan" to have name
# "Fulan".

db.disney_chars.find({name: "Mulan"}).
                forEach(function(doc) {
    {
        id = doc._id;
        new_name = doc.name.replace("Mulan", "Fulan");
        db.disney_chars.update({ _id : id},
                               {$set: {name: new_name}})
    };
});
```

Connection with PyMongo I

- To connect from within Python, we still have to **start a server**, and then in code we can do the following:

```
from pymongo import MongoClient

client = MongoClient() # Instantiate a client that will
                       # be connected to Mongo.
database = client['class_db'] # Create a variable holding
                              # a reference to the db you
                              # want to connect to.
collection = database['disney_chars'] # Create a variable
                                      # to hold the
                                      # collection you
                                      # want to connect to.
```

- Note that we access the database and/or collection by effectively using the *name* as a key (it's like indexing into a dictionary)

Connecting with PyMongo II

- It's also worth nothing that from within Python, connecting to a database or collection that doesn't exist works the same way as it does within the Mongo Shell - the database or collection is created if it doesn't already exist.

Issuing Queries with PyMongo

- Once we connect to Mongo using the `pymongo` library, we can enter queries almost like we did from the Mongo shell.

```
res = collection.find({'name': { '$ne': 'Mulan'}})
```

- Note this gives back a generator, which you can get results back from one at a time using `.next()`, or all at once using `all_res = list(res)`
- Basically, we kind of just have to wrap everything that we used in the Mongo Shell in quotes, unless it already was in quotes

Inserting and/or Updating

- We can `insert` or `update` by calling one of the following methods on the variable holding the collection:

```
# Methods to insert.
collection.insert_many()
collection.insert_one()

# Methods to update (allows the upsert argument)
collections.update_many()
collections.update_one()
```

Additional Mongo resources

- SQl to Mongo translator:
  - querymongo
- SQL to Mongo conversion guide:
  - Conversion Guide
- Aggregation Guide