NoSQL with MongoDB

Objectives

- Compare and contrast SQL and NoSQL
  - Explain use cases for NoSQL
  - Explain use cases for SQL
  - Discuss why we need both options
- Develop basic familiarity with MongoDB

SQL Review

- What is SQL?
  - Structured Query Language
- SQL allows us to interact with Relational Database Management Systems (RDBMS)
  - Model **relations** in the data
    - Stores data about one object across multiple tables
      - {student_id, course_id}, {student_id, student_name} -> join on student_id and filter on course_id for names of all students in a course
    - Query data and relations efficiently
  - Maintain data consistency and integrity

SQL Review

- Tables, Columns (fields), Rows (records)
  - Each column is of a certain data type
  - Each row is an entry in the table
    - It holds values for each one of the columns
  - Tables are specified by a **schema** that defines the structure of the data
    - We specify the schema ahead of time

NoSQL

- NoSQL
    - “Non SQL”, “Non relational”, “Not Only SQL”
    - *“Not Only SQL”* because stacks often use both NoSQL and SQL for different purposes
- Many NoSQL databases (though not all) are **document-oriented**
    - Each object (row/document) is stored in one place
    - Each object can be completely different from all others
- There is **no schema** (actually, there is, but it’s *implicit*)
    - Each document can have or not have whatever fields are appropriate for that particular document

MongoDB

- MongoDB is a document-oriented database, an alternative to RDBMS
- Used for storing semi-structured data
- JSON-like objects form the data model, rather than RDBMS tables
- No schema, No joins, No transactions
- Sub-optimal for complicated queries
- MongoDB is made up of databases which contain collections (analogous to tables)
- A collection is made up of documents (analogous to rows or records)
- Each document is made up of key-value pairs (analogous to columns)
- ***RDBMS defines columns at the table level, document oriented database defines its fields at a document level.***
- CURSOR: When you ask MongoDB for data, it returns a pointer to the result set called a cursor.
- Actual execution is delayed until necessary.

Roughly Analogous:

- NoSQL
  - MongoDB
    - Database
      - Collection
        - Field
          - Document

- SQL
  - PostgreSQL
    - Database
      - Table
        - Column
          - Row

- Spreadsheet
  - Excel
    - File
      - Sheet
        - Column
          - Row

SELECT id, user_id, status FROM people	db.people.find({ }, { user_id: 1, status: 1 })
SELECT user_id, status FROM people	db.people.find({ }, { user_id: 1, status: 1, _id: 0 })
SELECT * FROM people WHERE status = "A"	db.people.find({ status: "A" })
SELECT user_id, status FROM people WHERE status = "A"	db.people.find({ status: "A" }, { user_id: 1, status: 1, _id: 0 })
SELECT * FROM people WHERE status != "A"'	db.people.find({ status: { $ne: "A" } })
SELECT * FROM people WHERE status = "A" AND age = 50	db.people.find({ status: "A", age: 50 })
SELECT * FROM people WHERE status = "A" OR age = 50	db.people.find({ $or: [{ status: "A" } , { age: 50 }] })
SELECT * FROM people WHERE age > 25	db.people.find({ age: { $gt: 25 } })
SELECT * FROM people WHERE age < 25	db.people.find({ age: { $lt: 25 } })
SELECT * FROM people WHERE age > 25 AND age <= 50	db.people.find({ age: { $gt: 25, $lte: 50 } })
SELECT * FROM people WHERE user_id like "%bc%"	db.people.find({ user_id: /bc/ })

COMMAND HELPERS

The following table lists some common help commands which are available in the `mongo` shell:

`help`	Show help.
`db.help()`	Show help for database methods.
`db.<collection>.help()`	Show help on collection methods. The **`<collection>`** can be the name of an existing collection or a non-existing collection.
`show dbs`	Print a list of all databases on the server.
`use <db>`	Switch current database to **`<db>`**. The mongo shell variable db is set to the current database.
`show collections`	Print a list of all collections for the current database.
`show users`	Print a list of users for the current database.
`show roles`	Print a list of all roles, both user-defined and built-in, for the current database.
`show profile`	Print the five most recent operations that took 1 millisecond or more.
`show databases`	Print a list of all available databases.

FOR MORE INFORMATION

https://docs.mongodb.com/manual/reference/mongo-shell/

CRUD METHODS

Queries typically take the following form:

```
db.<collection>.<method>( <filter>, <options>)
```

`db` refers to the current database. **`<collection>`** is the name of the target collection for your query. For **`<method>`**, substitute the desired query method, (examples below). Each method has its own **`<options>`** for what it will do with the matching document(s).

`db.collection.insertOne()`	Inserts a document into a collection.
`db.collection.insertMany()`	Inserts multiple documents into a collection.
`db.collection.find()`	Selects documents in a collection based on the filter and returns a cursor to the selected documents.
`db.collection.updateOne()`	Updates a single document within the collection based on the filter.
`db.collection.updateMany()`	Updates all documents within the collection that match the filter.
`db.collection.replaceOne()`	Replaces a single document within the collection based on the filter.
`db.collection.deleteOne()`	Removes a single document from a collection based on the filter.
`db.collection.deleteMany()`	Removes all documents that match the filter from a collection.

FOR MORE INFORMATION

https://docs.mongodb.com/manual/crud/

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.

QUERY FILTER PARAMETERS AND WHAT THEY MATCH

MongoDB uses a key-value structure to create query filter parameters, which you can use in the `mongo` shell or with a driver in a client application. For example, the following query finds all documents in the collection named `inventory` in which the `qty` field contains a value greater than `10`:

```
db.inventory.find({ "qty" : { $gt: 10 }})
```

Queries take documents as query filter parameters, shown as examples below. Multiple filter parameters can be included in one document, separated by commas.

`{a: 10}`	Docs where **a** is **10** or an array containing the value **10**.
`{a: 10, b: "hello"}`	Docs where **a** is **10** and **b** is "**hello**".
`{a: {$gt: 10}}`	Docs where **a** is greater than **10**. *Also available:* ***$lt*** *(<),* ***$gte*** *(>=),* ***$lte*** *(<=), and* ***$ne*** *(!=).*
`{a: {$in: [10, "hello"]}}`	Docs where **a** is either **10** or "**hello**".
`{a: {$all: [10, "hello"]}}`	Docs where **a** is an array containing both **10** and "**hello**".

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)

FIELD UPDATE OPERATORS

`{$inc: {a: 2}}`	Increment **a** by **2**.
`{$set: {a: 5}}`	Set **a** to the value **5**.
`{$unset: {a: 1}}`	Delete the **a** key.
`{$max: {a: 10}}`	Set **a** to the greater value, either current or **10**. If **a** does not exist, set **a** to **10**.
`{$min: {a: -10}}`	Set **a** to the lowest value, either current or **-10**. If **a** does not exist, set **a** to **-10**.
`{$mul: {a: 2}}`	Set **a** to the product of the current value of **a** and **2**. If **a** does not exist set **a** to **0**.
`{$rename: {a: "b"}}`	Rename field **a** to **b**.
`{$setOnInsert: {a: 1}}, {upsert: true}`	Set field **a** to **1** in case of upsert operation.

`{$push: {a: {$each: [50, 60, 70], $position: 0}}}`	Insert **50**, **60**, and **70** starting at position **0** of the array **a**. *$position can only be used with the $each modifier.*
`{$addToSet: {a: 1}}`	Append the value **1** to the array **a** (if the value doesn't already exist).
`{$addToSet: {a: {$each: [1, 2]}}}`	Append both **1** and **2** to the array **a** (if they don't already exist).
`{$pop: {a: 1}}`	Remove the last element from the array **a**.
`{$pop: {a: -1}}`	Remove the first element from the array **a**.
`{$pull: {a: ($gt: 5}}}`	Remove all values greater than **5** from the array **a**.
`{$pullAll: {a: [5, 6]}}`	Remove multiple occurrences of **5** or **6** from the array **a**.

FOR MORE INFORMATION

http://docs.mongodb.org/manual/reference/operator/update/

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

AGGREGATION FRAMEWORK:

The aggregation pipeline, part of the MongoDB query language, is a framework for data aggregation modeled on the concept of data processing pipelines. Documents enter a multi-stage pipeline that transforms the documents into aggregated results. Pipeline stages appear in an array. Documents pass through the stages in sequence. Structure an aggregation pipeline using the following syntax:

```
db.<collection>.aggregate( [ { <stage1> }, { <stage2> } ... ]
)
```

COMMON AGGREGATION FRAMEWORK STAGES

`{$match: {a: 10}}`	Passes only documents where **a** is **10**.	Similar to find()
`{$project: { a: 1, _id:0}}`	Reshapes each document to include only field **a**, removing others.	Similar to find() projection
`{$project: { new_a: "$a"  }}`	Reshapes each document to include only **_id** and the new field **new_a** with the value of **a**.	{a:1} => {new_a:1}
`{$project: { a: {$add:["$a", "$b"]}}}`	Reshapes each document to include only **_id** and field **a**, set to the sum of **a** and **b**.	{a:1, b:10} => {a: 11}

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

What to grab
(SQL
WHERE)

GROUP BY and
AGGREGATION
function

- 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 } }
])
```

Whether to
SORT

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

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)