Databases for Data Science

Lecture 11 · 2022-10-10

The plan

Lectures		
Today	10/10	Aggregation framework; schema validation
Wednesday	10/12	Discuss other DBs; JSON in PostGreSQL; topics
After break Monday	10/24	Final Exam

Recap: Aggregation pipeline

We specify a list of **stages**.

- `stages[0]` output → `stages[1]` input, etc
- Each stage has a particular function.

```
coll = db.listingsAndReviews

coll.aggregate( [
  {
    '$group':
    {
      # Use _id to specify what we GROUP BY
      '_id': '$bedrooms',

      # Add other columns derived from each group
      'how_many': {'$sum': 1},
      'avg_bath': {'$avg': '$bathrooms'}}
  }
])
```

Equivalent SQL:

```
SELECT
  bedrooms AS _id,
  COUNT(*) AS how_many,
  AVG(bathrooms) AS avg_bath
FROM
  listingsAndReviews
GROUP BY
  bedrooms;
```

Recap: Aggregation pipeline

- `aggregate` is a function; this is what actually runs the query and returns a cursor.
- The argument to aggregate is a list of *stages*.
- Each stage is an *object*. It has one key that specifies the type of stage.
  - e.g., `{ '$group': {...}}`
- That key maps to a value which parametrizes the stage.
- Inside this specification, we write a `$` whenever we're referring to one of the *original* fields.
  - and also when using an operator.

```
coll.aggregate( [
  {
    '$group':
    {
      '_id': '$bedrooms',
      'how_many': {'$sum': 1},
      'avg_bath': {'$avg': '$bathrooms'}}
  }
])
```

Aggregation pipeline

```
coll.aggregate( [
  {
    '$group':
    {
      '_id': '$bedrooms',
      'how_many': {'$sum': 1},
      'avg_bath': {'$avg': '$bathrooms'}}
  }
])
```

See mongodb.com/docs/manual/reference/operator/aggregation-pipeline/ for a list of stage types.

Exercise: Use `$match` to find all apartments with more than two bedrooms.

Exercise: Use `$project` on this output to get the name, country code, and total room count.

Exercise: Use `$group` on this output to get the average number of total rooms per country code.

Matching documents by schema

We can search for documents that match a particular JSON format.

```
target_schema = {
  "required": ["name", "host"],
  "properties": {
    "address": {
      "bsonType": "object",
      "required": ["country_code"],
      "properties": {
        "country_code": {
          "$enum": ["US", "CA"]
        }
      }
    }
  }
}
db.listingsAndReviews.find({"$jsonSchema": target_schema})
```

Exercise: find all of the listings with a `weekly_price`.

Schema validation

What if we want to constrain our collection to *only* accept documents with that format?

This is where *validators* come in.

Schema validation

What if we want to constrain our collection to *only* accept documents with that format?

This is where *validators* come in.

```
db.create_collection("my_collection", validator={
    # Some filter logic
}
```

How to think of this:

- The validator is just like the filter in a `find()` call.
- If something wouldn't be returned by that `find()`, it wouldn't be allowed by the validator.
- If you call `my_collection.find(validator)`, you will get everything in `my_collection`.

Schema validation

Example:

```
validator = {
    "$jsonSchema": {
      "bsonType": "object",
      "title": "Specify values for a person",
      "required": ["name","ssn"],
      "properties": {
        "ssn": {
          "bsonType": "string",
          "description": "SSN must be a string"
        }
        "dob": {
          "bsonType": "date",
          "description": "Date of birth, if known, must be represented by the `date` type"
        }
      }
    }
}

db.create_collection("people",validator=validator)
```

Schema validation

Validators can be any Mongo expression.

```
db.create_collection("numbers", validator={
  "$and": [
    {
      "$jsonSchema": {
        "required": ["x","y"],
        "properties": {
          "x": {"bsonType": "int"},
          "y": {"bsonType": "int"}
        }
      }
    },
    {
      "$expr": {
        "$lt": ["$x", "$y"]
      }
    }
  ]
})
```

Exercise: write a validator for a `courses` collection. Each course must have a name and an *array of* instructors.

Exercise: write a validator for a `financial_aid` collection. Include mandatory fields for `student_id` and `scholarship`; require that scholarships are numbers between $0 and $50,000.

Aggregation: joins via `$lookup`

The `$lookup` operator allows us to perform a kind of pseudo-join.

```
<collection>.aggregate([{
  "$lookup": {
    "from": <other collection>,
    "localField": <field to match here>,
    "foreignField": <field to match there>,
    "as": <what to call the output>
  }
}])
```

```
db.bookings.aggregate([{
  "$lookup": {
    "from": "charges"
    "localField": "bookingnumber",
    "foreignField": "bookingnumber",
    "as": "charges_booked"
  }
}])
```

Exercise: Create a collection of professors and a collection of courses.

- Associate each with a department, e.g. "econ" or "cs".

Exercise: Insert some data of your own creation.

Exercise: Use `$lookup` to get the list of courses in each professor's department.

Discussion: Assignment 2

Let's go through the prompt together.