Announcements (Thu., Oct. 21)

- Homework 3 due in 1½ weeks
- Weekly project progress update due today (and every Thu.)
- Project milestone 3 due in 2 weeks

Announcements (Tue. Oct. 26)

- Homework 3 due in one week
- Weekly project progress update due (every) Thu.
- Project milestone 3 due in 1½ weeks
  - A short (≤ 5 min.) video showing a working (perhaps not complete) website interacting with the backend
  - A bigger sample database that “stress-test” efficiency and design
JSON (JavaScript Object Notation)

- Very lightweight data exchange format
- Much less verbose and easier to parse than XML
- Increasingly used for data exchange over Web; many Web APIs use JSON to return responses/results
- Based on JavaScript
  - Conforms to JavaScript object/array syntax—you can directly manipulate JSON representations in JavaScript
- But it has gained widespread support by all programming languages

Example JSON vs. XML

```
{  "ISBN": "ISBN-10",  "price": 80.00,  "title": "Foundations of Databases",  "authors": ["Abiteboul", "Hull", "Vianu"],  "publisher": "Addison Wesley",  "year": 1995,  "sections": [    {"title": "Section 1", "sections": [      {"title": "Section 1.1"},      {"title": "Section 1.2"}    ]},    {"title": "Section 2"}  ]}
```

```
<bibliography>
</book>
</bibliography>

JSON data model

- Two basic constructs
  - **Array**: comma-separated list of “things” enclosed by brackets
    - Order is important
  - **Object**: comma-separated set of pairs enclosed by braces; each pair consists of an attribute name (string) and a value (any “thing”)
    - Order is unimportant
    - Attribute names “should” be unique within an object
- Simple types: numbers, strings (in double quotes), and special values “true”, “false”, and “null”
- Thing = a simple value or an array or an object
**JSON Schema**

- Recall the advantages of having a schema
  - Defines a structure, helps catch errors, facilitates exchange/automation, informs optimization...
  - Just like relational data and XML, JSON is getting a schema standard too!
  - Up and coming, but still a draft at this stage

```json
{
  "definitions": {
    "sections": {
      "type": "array",
      "description": "Sections.",
      "sections": {
        "$ref": "#definitions/sections"
      }},
    "minItems": 0
  },
  "title": "Book",
  "type": "object",
  "properties": {
    "ISBN": {
      "type": "string",
    },
    "price": {
      "type": "number",
      "description": "The book's price."
    },
    "sections": {
      "$ref": "#definitions/sections"
    }
  }
}
```

---

**MongoDB**

- One of the “NoSQL” poster children
- Started in 2007
- Targeting semi-structured data in JSON
- Designed to be easy to “scale out”
- Good support for indexing, partitioning, replication
- Nice integration in Web development stacks
- Not-so-great support for joins (or complex queries) or transactions

---

**Inside a MongoDB database**

- Database = a number of “collections”
- Collection = a list of “documents”
- Document = a JSON object
  - Must have an _id attribute whose value can uniquely identify a document within the collection

☞ In other words, a database has collections of similarly structured “documents”
  - Much like tables of records, as opposed to one big XML document that contains all data
Querying MongoDB

- `find()` and `sort()`
  - Analogous to single-table selection/projection/sort
- “Aggregation” pipeline
  - With “stages” analogous to relational operators
  - Join, group-by, restructuring, etc.
- MapReduce:
  - Supports user-defined functions
  - We will save this topic until later in this course
  - We won’t cover syntax for creating/updating MongoDB databases in lecture
    - See “Help” of the course website and read the manuals!

Key features to look out for

- Queries written as JSON objects themselves!
  - Natural in some cases (e.g., for specifying conditions on subsets of attributes), but awkward/misleading in others
- Simple path expressions using the “dot notation”
  - Analogous to XPath “/”
- Arrays within objects
  - Work on nested array directly using constructs like dot-index notation, `$elemMatch`, `$map`, and `$filter`
  - Or “unnest” an array so its elements get paired with the owner object in turn for pipeline processing
    - A fundamental concept in working with nested data

Basic MongoDB `find()`

- All books
  - `db.bib.find()`
- Books with title “Foundations of Databases”
  - `db.bib.find( { title: "Foundations of Databases" } )`
- Books whose title contains “Database” or “database” and whose price is lower than $50
  - `db.bib.find( { title: /Database/, price: { $lt:50 } } )`
- Books with price between $70 and $100
  - `db.bib.find( { price: { $gte:70, $lte:100 } } )`
  - By the way, why wouldn’t the following work?
    - `db.bib.find( { price: { $gte:70, $lte:100 } } )`
- Books authored by Widom
  - `db.bib.find( { authors: "Widom" } )`
  - Note the implicit existential quantification
No general “twig” matching!

• Suppose for a moment publisher is an object itself, with attributes name, state, and country

• The following query won’t get you database books by US publishers:
  ```
  db.bib.find({
    title: /database/,
    publisher: { country: "US" }
  })
  ```

  • Instead, the condition on publisher is satisfied only if it is an object with exactly one attribute, and this attribute must be named country and has value “US”

  • What happens is that MongoDB checks the equality against `{ country: "US"}` as an object, not as a pattern!

More on nested structures

• Dot notation for XPath-like path expressions

  • Books where some subsection title contains “1.1”
    ```
    db.bib.find({
      "sections.sections.title": /1.1/ }
    })
    ```

  • Note we need to quote the expression

  • Again, if the expression returns multiple things, the condition only needs to hold for at least one of them

• Use $elemMatch to ensure that the same array element satisfies multiple conditions, e.g.:
  ```
  db.bib.find({
    "sections": {
      "$elemMatch": {
        title: /Section/,
        "sections.title": /1\.|1/}
    }
  })
  ```

• Dot notation for specifying array elements

  • Books whose first author is Abiteboul
    ```
    db.bib.find({ "authors.0": "Abiteboul" })
    ```

  • Note 0-based indexing; again, need to quote the expression

find() with projection and sorting

• List just the book prices and nothing else
  ```
  db.bib.find({
    price: { $exists: true },
    _id: 0, price: 1
  })
  ```

  • The (optional) second argument to find() specifies what to project: 1 means to return, 0 means to omit
    - _id is returned by default unless otherwise specified

• List books but not subsections, ordered by ISBN
  ```
  db.bib.find({}, {"sections.sections":0}).sort({ISBN:1})
  ```

  • Output from find() is further sorted by sort()

  • Aggregation pipelines (next) are better suited for constructing more complex output

☞ "Aggregation pipelines" (next) are better suited for constructing more complex output
MongoDB aggregation pipeline

• Idea: think of a query as performing a sequence of “stages,” each transforming an input sequence of JSON objects to an output sequence of JSON objects

• “Aggregation” is a misnomer: there are all kinds of stages
  • Selection ($match), projection ($project), sorting ($sort)
  • Much of which find() and sort() already do
  • Computing/adding attributes with generalized projection ($project/$addFields), unnesting embedded arrays ($unwind), and restructuring output ($replaceRoot)
  • Operators to transform/filter arrays ($map/$filter)
  • Join ($lookup)
  • Grouping and aggregation ($group)
  • Operators to aggregate (e.g., $sum) or collect into an array ($push)

The congress MongoDB database

• As in your Homework 3
• Two collections, people and committees
  • Each object in people is a legislator
    • roles = array of objects
  • Each object in committees is a committee
    • members = array of objects
    • subcommittees = an array of subcommittee objects, each with its own members array
    • Each member object's id field references a legislator _id
Selection/projection/sorting

Find Republican legislators, output only their name and gender, sort by name

```javascript
db.people.aggregate(
    {
        $match: {
            "roles.party": "Republican"
        },
        $project: {
            _id: false,
            name: true,
            gender: true
        },
        $sort: {
            name: 1
        }
    }
)
```

- `aggregate()` takes an array of stages
- Note again quoting the dot notation
- Note again the semantics of comparing a list of values: i.e., the query finds legislators who have ever served roles as Republicans

Generalized projection

Find Republican legislators, output their name, gender, and roles as an array of types (sen or rep)

```javascript
db.people.aggregate(
    {
        $match: {
            "roles.party": "Republican"
        },
        $addFields: {
            compact_roles: {
                $map: {
                    input: "$roles",
                    as: "role",
                    in: ""$$role.type"
                }
            }
        },
        $project: {
            _id: false,
            name: true,
            gender: true,
            roles: "compact_roles"
        }
    }
)
```

- Use ""\$xxx"" to tell MongoDB to interpret \$xxx as a field in the "current" object instead of just a string literal
- In \$map, as defines a new variable to loop over elements in the \$roles array. For each input element, \$map computes the \$expression and appends its value to the output array
- Use ""\$\$xxx"" to tell MongoDB that \$xxx is a new variable created during execution (as opposed to a field in the current object)

Unnesting and restructuring

Create a list of subcommittees: for each, simply display its name and the name of the committee it belongs to

```javascript
db.committees.aggregate(
    {
        $unwind: "$subcommittees"
    },
    {
        $replaceRoot: {
            newRoot: {
                committee: "$displayname",
                subcommittee: "$subcommittees.displayname"
            }
        }
    }
)
```

For each input committee, \$unwind loops over its subcommittees array, one element at a time, and outputs a copy of the committee object, with its subcommittees value replaced with this single element
Join

For each committee (ignore its subcommittees), display its name and the name of its chair

```javascript
db.committees.aggregate([{
  addFields: {
    chair_member: {
      $filter: {
        input: "$members",
        as: "member",
        cond: {$eq: ["$$member.role", "Chairman"]}
      }
    }
  }
}, {
  $lookup: {
    from: "people",
    localField: "chair_member.id",
    foreignField: "_id",
    as: "chair_person"
  }
}, {
  $project: {
    _id: false,
    name: "$displayname",
    chair: {
      $arrayElemAt: ["$chair_person.name", 0]
    }
  }
}])
```

- In `$lookup`, `localField` specifies the attribute in the current object whose value will be used for lookup
- From specifies the collection in which to look for joining objects
- `foreignField` specifies the attribute therein to be joined
- `$lookup` creates an attribute in the current object with the same specified as, and sets it value to an array holding all joining objects
- Non-equality joins are also possible, with more complex syntax

☞ Non-equality joins are also possible, with more complex syntax

Grouping and aggregation

- Count legislators by gender, and list the names of legislators for each gender

```javascript
db.people.aggregate([{
  group: {
    _id: "$gender",
    count: {$sum: 1},
    list: {$push: "$name"}
  }
}])
```

- The required `$_id` specifies the grouping expression, whose value becomes the identifying attribute of output objects (one per group)
- Other attributes hold aggregate values, computed using aggregation operators
- `$sum` compute a total by adding each input
- `$push` creates an array by appending each input

Summary and discussion

- JSON is like a lightweight version of XML
  - But perhaps not as good for mixed contents
- Writing queries in JSON format is sometimes convenient, but confusing in many situations
  - Query as as pipeline: less declarative, but arguably easier to implement (especially to parallelize)
  - Reminds you of relational algebra?
- Nested structures requires more query constructs
  - `$map` stage, `$elemMatch`/`$map`/`$filter`/`$push`/`$arrayElemAt` operators, etc.
  - Distinction between the top-level and nested arrays is annoying
    - E.g., `$match` stage and `$filter` operator basically do the same thing
    - `$XQuery` is much nicer in this regard (with ability to nest queries in return)
- There is actually XQuery-like language for JSON called "JSONiq," but it remains less known