Project Sprint 2 (d2)

Campus Explorer Enhanced Query Service

Sprint 1 built a query engine to answer queries about UBC course sections. This deliverable will extend the input data to include data about the physical spaces where classes are held on campus. Also, the built query language was fairly simple, as you could not construct queries that would let you aggregate and compute values on the results of queries. In other words, the query engine returned data on a section-by-section basis. This deliverable will expand the query engine to enable result computation (e.g., to figure out the average for a *course* or figure out the number of seats for a *building*).

This deliverable will extend your d1 solution so you must continue to work with the same partner using the same repository. You will not have to hand anything in, though you will need to complete the open response assessment on the edX platform to receive your final deliverable grade. The deliverable is worth 25% of your final capstone grade. The ability to start on d3 will unlock once you achieve 60% or more on d2.

Dataset

This data has been obtained from the UBC Building and classrooms listing. The data is provided as a zip file: inside of the zip you will find `index.xml` which specifies each building on campus. The links in the `index.xml` link to files also in the zip containing details about each building and its rooms in XML format.

The dataset file can be found here sdmm.rooms.1.0.zip.

Checking the validity of the dataset

- Each XML file other than `index.xml` represents a building and its rooms.
- All buildings linked from the index should be considered valid buildings.
    - A valid building will always be in XML format.
    - Buildings not linked in the index should not be processed.

Additionally, the following assumptions do not require validation:

- There is a single `index.xml` file per dataset
- Regardless of paths, there is a single file per building code, i.e. there will never be two files with the same name but in different paths
- All XML elements and attributes associated with target data will be present in the same forms as in the provided zip file.

Reading and Parsing the Dataset

As with Deliverable 1, you will need to parse valid input files into internal objects or other data structures. You are not allowed to store the data in a database. You must also persist (cache) the model to disk for quicker access. Do not commit this cached file to version control, or AutoTest will fail in unpredictable ways.

There is a provided package called `parse5` that you should use to parse the XML files using the `parseFragment` method. We will call `addDataset` with `rooms` for the `InsightDatasetKind`. Parse5 also has an online playground where you can see the structure of a Document, which is the output of a parsed XML file. You must traverse this document in order to extract the building/rooms.

Finding the geolocation of each building

In addition to parsing the XML files, you must encode buildings' addresses to a latitude/longitude pair. This is usually performed using online web services. To avoid problems with our spamming different geolocation providers we will be providing a web service for you to use for this purpose. To obtain the geolocation of an address, you must send a GET request (using the `http` module) to:

```
http://sdmm.cs.ubc.ca:11316/api/v1/team<YOUR-TEAM-NUMBER>/<ADDRESS>
```

Where `ADDRESS` should be the URL-encoded version of an address (e.g., `6245 Agronomy Road V6T 1Z4` should be represented as `6245%20Agronomy%20Road%20V6T%201Z4`). Addresses should be given *exactly* as they appear in the data files, or `404` will be returned.

The response will match the following interface (either you will get `lat` & `lon` or `error`, but never both):

```
interface GeoResponse {
    lat?: number;
    lon?: number;
    error?: string;
}
```

Since we are hosting this service it *could* be killed by DOS attacks, please try not to overload the service. You should only need to query this when you are processing the initial dataset, not when you are answering queries.

To handle the requests, you must use the `http` package. If you try to update your project with other third party packages AutoTest will fail in unpredictable ways.

Query Engine

Regarding the query engine, the primary objective of this deliverable is twofold: (i) extend the query language to accommodate queries to a new dataset, i.e. `rooms`; and (ii) enable more comprehensive queries about the datasets, i.e. aggregate results.

At a high level, the new functionality adds:

- `GROUP`: Group the list of results into sets by some matching criteria.

- `APPLY` : Perform calculations across a set of results.
    - `MAX` : Find the maximum value of a field. For numeric fields only.
    - `MIN` : Find the minimum value of a field. For numeric fields only.
    - `AVG` : Find the average value of a field. For numeric fields only.
    - `SUM` : Find the sum of a field. For numeric fields only.
    - `COUNT` : Count the number of unique occurrences of a field. For both numeric *and* string fields.
- `SORT` : Order results on one or more columns.
    - You can sort by a single column as in D1, e.g., `"ORDER": "courses_avg"` ; or
    - You can sort using an object that directly specifies the sorting order (see query example)
        - `"dir": "UP"` : Sort results ascending.
        - `"dir": "DOWN"` : Sort results descending.
        - `"keys": ["courses_avg"]` : sorts by a single key
        - `"keys": ["courses_year", "courses_avg"]` : sorts by multiple keys
            - In this example the course average should be used to resolve ties for courses in the same year
- An empty (`{}`) `WHERE` clause signals that all rows should be returned.

EBNF

```
QUERY  ::= QUERY_NORMAL || QUERY_AGGREGATE

FILTER ::= 'find all entries' || 'find entries whose ' + (CRITERIA || (CRITERIA +
((' and ' || ' or ') + CRITERIA)*)

QUERY_NORMAL ::= DATASET + ', ' + FILTER + '; ' + DISPLAY(+ '; ' + ORDER)? + '.'
DATASET      ::= 'In ' + KIND + ' dataset ' + INPUT
DISPLAY      ::= 'show ' + KEY (+ MORE_KEYS)?
ORDER        ::= 'sort in ' + ('ascending ' || 'descending ')? + 'order by ' + KEY
(+ MORE_KEYS)?

QUERY_AGGREGATE ::= DATASET_GROUPED + ', ' + FILTER + '; ' + DISPLAY_GROUPED(+ '; '
+ ORDER_GROUPED)? + '.'
DATASET_GROUPED ::= DATASET + ' grouped by ' + KEY (+ MORE_KEYS)?
DISPLAY_GROUPED ::= 'show ' + KEY_C (+ MORE_KEYS_C)? + (', ' + AGGREGATION)?
ORDER_GROUPED   ::= 'sort in ' + ('ascending ' || 'descending ')? + 'order by ' +
KEY_C (+ MORE_KEYS_C)?
AGGREGATION     ::= 'where ' + AGG_DEF (+ MORE_DEFS)*

AGG_DEF    ::= INPUT + ' is the ' + AGGREGATOR + ' of ' + KEY
MORE_RULES ::= ((', ' + AGG_DEF +)* ' and ' + AGG_DEF)
AGGREGATOR ::= 'MAX' || 'MIN' || 'AVG' || 'SUM'

CRITERIA   ::= M_CRITERIA || S_CRITERIA
M_CRITERIA ::= M_KEY + M_OP + NUMBER
S_CRITERIA ::= S_KEY + S_OP + STRING

NUMBER   ::= [0-9] + (.)? + [0-9]
STRING   ::= '"' + [^*"]* + '"' // any string without * or " in it, enclosed by
double quotation marks

RESERVED ::= KEYWORD || M_OP || S_OP || AGGREGATOR || KIND
KEYWORD  ::= 'in' || 'dataset' || 'find' || 'all' || 'show' || 'and' || 'or' ||
'sort' || 'by' || 'entries' || 'grouped' || 'where' || 'is' || 'the' || 'of' ||
'whose'
M_OP     ::= 'is ' + ('not ' +)? ('greater than ' || 'less than ' || 'equal to ') +
NUMBER
S_OP     ::= (('is ' + ('not ' +)?) || (('includes ' || 'does not include ') ||
('begins' || 'does not begin' || 'ends' || 'does not end' + ' with '))) + STRING
KIND     ::= 'courses' || 'rooms'

INPUT    ::= string of one or more characters. Cannot contain spaces, underscores or
equal to RESERVED strings

KEY   ::= M_KEY || S_KEY
KEY_C ::= KEY || INPUT
MORE_KEYS   ::= ((', ' + KEY +)* ' and ' + KEY)
MORE_KEYS_C ::= ((', ' + KEY_C +)* ' and ' + KEY_C)
M_KEY ::= 'Average' || 'Pass' || 'Fail' || 'Audit' || 'Latitude' || 'Longitude' ||
'Seats' || 'Year'
S_KEY ::= 'Department' || 'ID' || 'Instructor' || 'Title' || 'UUID' || 'Full Name'
|| 'Short Name' || 'Number' || 'Name' || 'Address' || 'Type' || 'Furniture' ||
'Link'
```

Syntactic Checking (Parsing)

In addition to the syntactic checking from Sprint 1, you must check:

- `APPLY` keys are not allowed to contain the `_` character.

Semantic Checking

In addition to the semantic checking from Sprint 1, you must perform the following semantic check:

- The `key` in `APPLY` should be unique (no two `APPLY` targets should have the same name).
- If a `GROUP` is present, all `COLUMNS` terms must correspond to either `GROUP` terms or to terms defined in the `APPLY` block.
- `'SORT'` - Any keys provided must be in the `COLUMNS`. Otherwise, the query is invalid.
- `MAX/MIN/AVG` should only be requested for numeric keys.
- `WHERE` is completely independent of `GROUP` / `APPLY`

Other JS/Typescript considerations

- Ordering should be according to the `<` operator in TypeScript/JavaScript, not by `localeCompare` or the default `sort()` implementation.
- `AVG` should return a number rounded to two decimal places.
  Supporting `AVG` requires some extra challenges compared to the other operators. Since JavaScript numbers are represented by floating point numbers, performing this arithmetic can return different values depending on the order the operations take place. To account for this, you must use the `Decimal` package (already included in your package.json) taking the following protections.
  1. Convert you values to decimals (e.g., `new Decimal(number)`).
  2. Add the numbers being averaged (e.g., generate `total`).
  3. Calculate the average (`var avg = total.toNumber() / numRows`).
  4. Trim the average to the second decimal digit (`var res = Number(avg.toFixed(2))`)
- `SUM` should return a number rounded to two decimal places.
- `COUNT` should return whole numbers.
- `MIN/MAX` should return the same number that is in the originating dataset.

Valid keys

In addition to the valid keys from Sprint 1, this deliverable adds a variety of new keys. A valid query will not contain keys from more than one dataset (i.e. only courses_xx keys or only rooms_xx keys, never a combination).

If the `id` sent by the user is `rooms`, then the queries you will run will be using the following keys:

- rooms_fullname: `string`; Full building name (e.g., "Hugh Dempster Pavilion").
- rooms_shortname: `string`; Short building name (e.g., "DMP").
- rooms_number: `string`; The room number. Not always a number, so represented as a string.
- rooms_name: `string`; The room id; should be `rooms_shortname`+"_"+`rooms_number`.
- rooms_address: `string`; The building address. (e.g., "6245 Agronomy Road V6T 1Z4").
- rooms_lat: `number`; The latitude of the building. Instructions for getting this field are below.
- rooms_lon: `number`; The longitude of the building, as described under finding buildings' geolocation.
- rooms_seats: `number`; The number of seats in the room.
- rooms_type: `string`; The room type (e.g., "Small Group").
- rooms_furniture: `string`; The room type (e.g., "Classroom-Movable Tables & Chairs").
- rooms_href: `string`; The link to full details online (e.g., "http://students.ubc.ca/campus/discover/buildings-and-classrooms/room/DMP-201").

We are also adding one field to the `courses` dataset:

- courses_year: `number`; This key represents the year the course was offered. If the `"Section"` property is set to `"overall"`, the year for that section should be set to 1900.

Aggregation (step by step)

`GROUP: [term1, term2, ...]` signifies that a group should be created for every unique set of all N-terms (e.g., `[courses_dept, courses_id]` would have a group for every unique pair of department/id records in the dataset. Every member of a `GROUP` always matches all terms in the `GROUP` array.

As an example, suppose we have the following `courses` dataset (for the sake of simplicity, some keys are omitted):

```
[
    { "courses_uuid": "1", "courses_instructor": "Jean",  "courses_avg": 90,
"courses_title" : "310"},
    { "courses_uuid": "2", "courses_instructor": "Jean",  "courses_avg": 80,
"courses_title" : "310"},
    { "courses_uuid": "3", "courses_instructor": "Casey", "courses_avg": 95,
"courses_title" : "310"},
    { "courses_uuid": "4", "courses_instructor": "Casey", "courses_avg": 85,
"courses_title" : "310"},
    { "courses_uuid": "5", "courses_instructor": "Kelly", "courses_avg": 74,
"courses_title" : "210"},
    { "courses_uuid": "6", "courses_instructor": "Kelly", "courses_avg": 78,
"courses_title" : "210"},
    { "courses_uuid": "7", "courses_instructor": "Kelly", "courses_avg": 72,
"courses_title" : "210"},
    { "courses_uuid": "8", "courses_instructor": "Eli",   "courses_avg": 85,
"courses_title" : "210"},
]
```

We want to query the dataset to aggregate courses by their title and obtain their average. Our aggregation query would look like this:

```
{
 "WHERE": {
        "GT": { "courses_avg": 70 }
 },
 "OPTIONS": {
  "COLUMNS": ["courses_title", "overallAvg"]
 },
 "TRANSFORMATIONS": {
  "GROUP": ["courses_title"],
  "APPLY": [{
   "overallAvg": {
    "AVG": "courses_avg"
   }
  }]
 }
}
```

For this query, there are two groups: one that matches `"courses_title" = "310"` and other that matches `"210"`. You need to somehow have an intermediate data structure that maps our entries to their matched groups:

```
310 group =
[
    { "courses_uuid": "1", "courses_instructor": "Jean",  "courses_avg": 90,
"courses_title" : "310"},
    { "courses_uuid": "2", "courses_instructor": "Jean",  "courses_avg": 80,
"courses_title" : "310"},
    { "courses_uuid": "3", "courses_instructor": "Casey", "courses_avg": 95,
"courses_title" : "310"},
    { "courses_uuid": "4", "courses_instructor": "Casey", "courses_avg": 85,
"courses_title" : "310"},
]

210 group =
[
    { "courses_uuid": "5", "courses_instructor": "Kelly", "courses_avg": 74,
"courses_title" : "210"},
    { "courses_uuid": "6", "courses_instructor": "Kelly", "courses_avg": 78,
"courses_title" : "210"},
    { "courses_uuid": "7", "courses_instructor": "Kelly", "courses_avg": 72,
"courses_title" : "210"},
    { "courses_uuid": "8", "courses_instructor": "Eli",   "courses_avg": 85,
"courses_title" : "210"},
]
```

The last step is fairly simple, we execute the apply operation in each group. Hence, the average of 310 is `(90 + 80 + 95 + 85)/4 = 87.5` whereas for the second group the average is `77.25`. Our final result for the previous query would be:

```
[
    { "courses_title" : "310", "overallAvg": 87.5},
    { "courses_title" : "210", "overallAvg": 77.25},
]
```

Notice that we can have more elaborate groups such as discovering if a specific instructor of a course has a better average than other instructors (i.e., `"GROUP": ["courses_instructor", "courses_title"]`). In that cause, we would have four groups `310 - Jean`, `310 - Casey`, `210 - Kelly`, and `210 - Eli`. Once again, the average operation would be executed for entries that match each group.

Note that there are different data structures that can be used to store your groups. Feel free to use whatever better suits you.

Query examples

```
In rooms dataset rooms, find entries whose Short Name is \"ESB\" and Seats is
greater than 100; show Short Name, Number and Seats; sort in ascending order by
Seats.

In courses dataset courses grouped by Department, find entries whose Department is
\"cpsc\"; show Department, and avgGrade, where avgGrade is the AVG of Average.
```

API

There are no changes in the API for this deliverable, it is the same as the one in Sprint 1.

Testing

There are no changes in the testing instructions, they are the same as in Sprint 1. To test D2, call AutoTest with `@ubcbot #d2`.

Getting started

There is no best way to get started, but you can consider each of these in turn. Some possible options that could be pursued in any order (or skipped entirely):

- Start by looking at the data file we have provided and understanding what kind of data you will be analyzing and manipulating. It is crucial to understand that `index.xml` and the other files have different structures. You will need to extract different, though complementary information, from each one of them.
- Get some buildings' addresses and ignoring the dataset parsing, write tests to get the geolocation of these addresses.
- Look at the sample queries in the deliverable description. From these queries, figure out how you would want the data arranged so you can answer these queries.
- Ignoring the provided data, create a fake dataset with few entries. Write the portion of the system that would perform the `GROUP` and `APPLY` operations in this small dataset.

Trying to keep all of the requirements in mind at once is going to be overwhelming. Tackling a single task that you can accomplish in an hour is going to be much more effective than worrying about the whole deliverable at once. Iteratively growing your project from small task to small task is going to be the best way to make forward progress.

Assessment

Please refer to the Project Description for more information on grading.