BDS Query

23 сентября 2015 г. 17:08

Purpose

This document describes how to formulate queries against private and public data in the Biogeochemistry Data System (BDS): First through the User Interface and then as text documents that can be used programmatically (i.e. not in the User Interface but by other programs).

Contents

Case Study DMSP DHPS

BDS UI Query Syntax

JSON Query Syntax

Queries are encapsulated in a JSON file called **query.json**. JSON stands for JavaScript Object Notation. It is documented at http://json.org and from this page we begin by quoting:

> **JSON** (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.
>
> JSON is built on two structures:
>
> - A collection of name/value pairs. In various languages, this is realized as an *object*, record, struct, dictionary, hash table, keyed list, or associative array.
> - An ordered list of values. In most languages, this is realized as an *array*, vector, list, or sequence.
>
> These are universal data structures. Virtually all modern programming languages support them in one form or another. It makes sense that a data format that is interchangeable with programming languages also be based on these structures.

The query file is built using syntax from braces, colons, double-quote string delimiters, commas and indentation. Here is the basic row-query example for BDS:

```
{
    "queryType": "RowQuery",
    "region": { "type": "Any" },
    "time": { "type": "Any" },
    "datasets": { "type": "Any" },
    "targetTable": { "type": "MS", "level": "1.3", "tableName": "MS-1.3" },
    "rowFilter": "mean_mass >= 315 && mean_mass <= 320"
}
```

Notice that the key:value pair construction is maintained throughout; where the values may themselves be embedded JSON. The outer encapsulation permits any number of key:value pairs, in this case six; and the indentation is entirely to help make the query readable.
Before going further it is worth asking 'How does this JSON query arrive at the BDS and how does a table of results make its way back?' Answer: The TPQ program interprets the **query.json** file and carries out a send/receive transaction with BDS over the web. The result is a CSV table called **results.csv**.

New:

Understanding Queries

First things first - before writing the first query one needs to understand what BDS can be queried for. The answer is: BDS can provide one of the following:

1. Identifiers of all the datasets fulfilling some criteria
2. Identifiers of tables fulfilling some criteria from datasets fulfilling some other criteria
3. Rows fulfilling some criteria from tables fulfilling some other criteria from datasets fulfilling some more criteria

Mentioned criteria include:

1. For datasets - location of samples (existence of at least one sample taken in designated location), date of sample collection (again, existence of at least on sample collected during designated period of time), and ownership (private, public, belonging to a certain list)
2. For tables - type, level, and name inside the level where the table resides in the dataset (e.g. MS, 1.1, MS-1.1-supp), satisfaction of the table predicate (e.g. max(m) <= 500)
3. For rows - satisfaction of the row predicate (e.g. formula == "C22H38O12"

Please, refer to the corresponding section for the complete explanation on what sorts of predicates BDS can ingest. Here're a few examples of requests, BDS can process:

1. What are the datasets in which there are carboys registered with lat > 50 deg North
2. Give me the masspec results for the formula C22H38O12 across all public samples (Level 1.4)
3. Same as 2 but from a list of 30 formulas
4. Give me the masspec results (1.4) for molecules with mass between 200 and 280

Writing Queries

As it was already mentioned, queries are passed to TPQ inside a JSON file. Inside that file there's one JSON object (that means, that contents of the file should be encapsulated in a pair of curly braces) with up to 7 fields containing all the information about the query. These fields are:

1. queryType - tells TPQ what BDS should return: datasets, tables, or rows
2. region - tells system to consider only datasets with samples collected within provided geographical region
3. time - tells system to consider only datasets with samples collected during provided period of time
4. datasets - restricts considered datasets either by ownership or by explicit list
5. targetTable - tells system which table on which level of which type it should consider (only for table and row queries)
6. tableFilter - allows to specify a predicate which considered tables will have to satisfy (only for table and row queries)
7. rowFilter - allows to specify a predicate which output rows will have to satisfy (only for row queries)

So, contents of "query.json" have the following form:

```
{
    "queryType": "<query type>",
    "region": { <spatial restriction> },
    "time": { <temporal restriction> },
```

```
    "datasets": { <dataset restriction> } [,
    "targetTable": { <target table path> } [,
    "tableFilter": "<table predicate>"] [,
    "rowFilter": "<row predicate>" ] ]
}
```

Here required parameters are enclosed in angular braces ("<>") and optional (required only for some types of queries) parts are enclosed in square braces ("[]"). Let's go over all the fields and explain them in detail.

queryType

Value `<query type>` of the field queryType should be

- DataSetQuery for queries that should return IDs of datasets
- TableQuery for queries that should return IDs of tables within datasets
- RowQuery for queries that should return data rows

region

Value of this field is a JSON object (as can be seen by curly braces). It has the following form:
`<spatial restriction> = "type": "<Any|Box>" [, "box": { "latMin": <number>, "latMax": <number>, "lonMin": <number>, "lonMax": <number> } ]`
"type" is "Any" if no geographical restrictions on considered datasets are required, or "Box" if considered datasets should be limited to those containing at least one sample collected within a provided rectangular geobox. In the latter case, parameter "box" also has to be provided. It's an object with fields containing minimum and maximum values for longitude and latitude of place where the sample should have been collected. `<number>` is a double-precision floating point number (without quotes). E.g.:

- `"region": { "type": "Any" }`
- `"region": { "type": "Box", "box": { "latMin": 29.0, "latMax": 30.0, "lonMin": 84.0, "lonMax": 85.0 } }`

time

Value of this field is a JSON object (as can be seen by curly braces). It has the following form:
`<temporal restriction> = "type": "<Any|Range>" [, "range": { "start": "<timestamp>", "end": "<timestamp>" } ]`
"type" is "Any" if no temporal restrictions on considered datasets are required, or "Range" if considered datasets should be limited to those containing at least one sample collected within a provided time period. In the latter case, parameter "range" also has to be provided. It's an object with fields containing timestamps for start and end of the time period, during which the sample should have been collected. `<timestamp>` is a timestamp of the form "YYYY-MM-DDTHH:mm:ss". E.g.:

- `"time": { "type": "Any" }`
- `"time": { "type": "Range", "range": { "start": "2014-01-02T00:00:00", "end": "2014-01-05T00:00:00" } }`

datasets

Value of this field is a JSON object (as can be seen by curly braces). It has the following form:
`<dataset restriction> = "type": "<Any|My|Public|CompleteList>" [, "list": <JSON array of IDs> ]`
This field allows to further limit the set of considered datasets - either by ownership or by residence in an explicitly provided list. "type" should be one of the following

- "Any" if no additional restrictions are required
- "My" if only datasets owned by you should be considered
- "Public" if only publicly available datasets should be considered
- "CompleteList" if considered datasets should be limited to an explicitly provided list

In the last case, parameter "list" has to be provided. It has to be a square-bracket enclosed comma-separated list of dataset identifiers each encapsulated in double quotes, e.g.:

```
"list": [ "900beeaf-681b-4513-95bb-b7ad20cdf0f0",
"7a9e2a3d-5e55-4c3f-8945-1808915fef96" ]
```
Here're a few complete examples:

- `"datasets": { "type": "Any" }`
- `"datasets": { "type": "My" }`
- `"datasets": { "type": "Public" }`
- `"datasets": { "type": "Any", "list": [ "900beeaf-681b-4513-95bb-b7ad20cdf0f0", "7a9e2a3d-5e55-4c3f-8945-1808915fef96" ] }`

targetTable

This field is required only for table and row queries. It tells the system, which table within a dataset user is interested in. Datasets that do not contain such table will be ignored. This field holds no meaning for dataset queries.

Value of this field is a JSON object (as can be seen by curly braces). It has the following form:

```
<target table path> = "type": "<type>", "level": "<level>", "tableName":
"<table>"
```

"type" is name of the TYPE in BDS, such as "MS" or "EEM". "level" is the name of the level of that TYPE, e.g. "1.0", "1.3". "tableName" is the name of the table within a level (there may be more than one table on any given level, e.g. level 1.1 of MS type has 4 tables: "MS-1.1", "MS-1.1-supp", "MS-1.1-2", "MS-1.1-2-supp").

E.g.:

- `"targetTable": { "type": "MS", "level": "1.3", "tableName": "MS-1.3" }`
- `"targetTable": { "type": "EEM", "level": "2.0", "tableName": "EEM-2.0-Ex" }`

tableFilter and rowFilter

Two most interesting fields. Their values are predicates, which, respectively, output tables and rows will have to satisfy. This allows to put custom content-based criteria into queries. For table queries rowFilter should not be present, for row queries tableFilter may be present, and if so only data from tables satisfying it will be output.

The aim of this section is to explain, how to write such predicates. Let's begin with row predicates as table predicates can contain them as their parts.

Row predicate is a logic function of a table row, which for that purpose can be considered a record with fields corresponding to columns of that table. That function has to be presented as a logical expression consisting of comparisons of field values to constants and each other joined with logic operators (and/or/not). Usage of parentheses to explicitly point out evaluation order is allowed. In that expression field values can be accessed by the name of the field, if that name does not contain whitespaces or other special characters (broadly speaking, it can be said that fields can be accessed by their names if such names can be used as identifiers in C-like languages), or they can be accessed via the following construction:

`field(<string>)`

Where <string> is a sting literal as defined in C# language spec (string encapsulated in double quotes where special characters are represented with escape-sequences, such as "\n" for line break, "\"" for double quote, "\\" for slash).

Row predicates can be strictly defined using BNF (see https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_Form):

```
<row predicate> ::= "(" <row predicate> ")"
                  | "!" <row predicate>
                  | <row predicate> "&&" <row predicate>
                  | <row predicate> "||" <row predicate>
                  | <row condition>
```

```
<row condition> ::= <field> <relation> <field>
                  | <field> <relation> <number>
                  | <number> <relation> <field>
                  | <field> <eq relation> <string>
                  | <string> <eq relation> <field>

<field> ::= <identifier> | "field("<string>")"

<eq relation> ::= "==" | "!="

<relation> ::= <eq relation> | ">" | "<" | ">=" | "<="
```

Here <identifier>, <number>, and <string> are defined respectively as identifier, numeric literal, and string literal in C# sense. "!" means logical "not", "&&" means logical "and", and "||" means logical "or". Logical operator precedence is "!", then "&&", then "||". Operators "&&" and "||" are left associative and short-circuit. Here's an example of a row predicate:

```
field("a@\u000f") > -3.e+1 || mass <= 300 && !formula == "C2H5OH"
```

It's satisfied only by rows where value of field "a@ " (last symbol is a symbol with UTF-16 code 000f) is greater than -30 or by rows where both value of field "mass" is less than or equal to 300 and value of field "formula" is not equal to "C2H5OH".

Now let's turn to table predicates. Table predicate is a logic function of a table, which is viewed either as a set of rows or as a set of columns depending on what we want to check. Inside table predicate we can check if

- All rows of a table satisfy some row predicate
- There exist a row in the table that satisfies some row predicate
- A certain (in)equality that may contain averages, minimums, or maximums of values of table columns and/or constants is true
- Table contains a column with given name
- Table satisfies a logic function of the above

Here's a rather self-explanatory BNF that explains how the above is achieved:

```
<table predicate> ::= "(" <table predicate> ")"
                    | <table predicate> "&&" <table predicate>
                    | <table predicate> "||" <table predicate>
                    | "!" <table predicate>
                    | <terminal table predicate>

<terminal table predicate> ::= <quantifier> "(" <row predicate> ")"
                             | <numeric field reduction> <relation> <numeric
                           field reduction>
                             | <numeric field reduction> <relation> <number>
                             | <number> <relation> <numeric field reduction>
                             | <boolean field reduction>

<quantifier> ::= "all" | "any"

<numeric field reduction> ::= <numeric field function>  "(" <field> ")"

<numeric field function> ::= "avg" | "min" | "max"

<boolean field reduction> ::= <boolean field function>  "(" <field> ")"

<boolean field function> ::= "exists"
```

Here `<field>` and `<row predicate>` are defined above in the part concerning row predicates. Here's an example of table predicate:

```
any(formula == "C2H5OH") && (avg(mass) > min(mass)) || exists(temp)
```

Such predicate is satisfied by tables containing column "temp" and tables which contain formula "C2H5OH" and a numeric column "mass", average value of which is greater than its minimum value.

At last, here's what happens if something goes wrong during predicate evaluation:

Case	Execution Result
Comparisons with non-existent fields/columns	Always result in 'false'.
Column type mismatch	All such comparisons result in 'false'.
Table column reduction results in exception	Comparison with that reduction results in 'false'.

Examples

Example 1. Get IDs of all public datasets containing at least one sample taken in a place with latitude between 29 and 30 and longitude between 84 and 85 and at least one sample (possibly the same one) taken between 2-Jan-2014 and 5-Jan-2014:

```
{
    "queryType": "DataSetQuery",
    "region": { "type": "Box", "box": { "latMin": 29.0, "latMax": 30.0,
"lonMin": 84.0, "lonMax": 85.0 } },
    "time": { "type": "Range", "range": { "start": "2014-01-02T00:00:00",
"end": "2014-01-05T00:00:00" } },
    "datasets": { "type": "Public" }
}
```

Example 2. Get IDs of all MS 1.3 tables and datasets containing them such that

- Dataset containing the table contains at least one sample taken in a place with latitude between 29 and 30 and longitude between 84 and 85
- Dataset containing the table belongs to user requesting the query
- MS 1.3 table contains a row with formula "C10H9O4"

```
{
    "queryType": "TableQuery",
    "region": { "type": "Box", "box": { "latMin": 29.0, "latMax": 30.0,
"lonMin": 84.0, "lonMax": 85.0 } },
    "time": { "type": "Any" },
    "datasets": { "type": "My" },
    "targetTable": { "type": "MS", "level": "1.3", "tableName": "MS-1.3" },
    "tableFilter": "any(Formula==\"C10H9O4\")"
}
```

Example 3. Get all rows (with IDs of tables and datasets containing them attached) from MS 1.3 tables such that

- Dataset containing the row contains at least one sample taken between 2-Jan-2014 and 5-Jan-2014
- Table containing the row has a row (possibly another one) with formula "C10H9O4"
- Value of "mean_mass" is between 315 and 320

```
{
    "queryType": "RowQuery",
    "region": { "type": "Any" },
    "time": { "type": "Range", "range": { "start": "2014-01-02T00:00:00",
"end": "2014-01-05T00:00:00" } },
    "datasets": { "type": "Any" },
    "targetTable": { "type": "MS", "level": "1.3", "tableName": "MS-1.3" },
    "tableFilter": "any(Formula==\"C10H9O4\")",
```

```
    "rowFilter": "mean_mass >= 315 && mean_mass <= 320"
}
```