HCA DCP Query Service

Dec 03, 2019

Contents

CHAPTER 1

Introduction

The HCA DCP Query Service provides an interface for scientists and developers to query metadata associated with experimental and analysis data stored in the Human Cell Atlas Data Coordination Platform (DCP). Metadata from the DCP Data Store are indexed and stored in an AWS Aurora PostgreSQL database.

Queries to the database can be sent over HTTP through the Query Service API Swagger Documentation or via the Query Builder.

1.1 Executing Queries

To execute a query via the swagger interface:

- Click on the green `/query` row to expand it
- Click `Try it out`
- Edit the request body by adding in your query and any parameters.

For example

```
{
  "params": {"s": 0},
  "query": "select * from files where size > %(s)s limit 10"
}
```

- click `Execute`

Note: Queries executed via the swagger interface can not contain new lines or tabs (you will need to edit some of the example queries by removing line breaks/tabs to make them work). These will generate a 400 error response containing the line `"detail": "Request body is not valid JSON"`. For complex queries that benefit from multi-line formatting we recommend using the Query Builder.

1.2 Async Queries

For long-running queries (runtime over 20 seconds), the Query Service supports asynchronous tracking of query results. When a long-running query triggers this mode, the caller will receive a `301 Moved Permanently` response status code with a `Retry-After` header. The caller is expected to wait the specified amount of time before checking the redirect destination, or use the query job ID returned in the response JSON body to check the status of the query job. The caller may turn off this functionality (and cause the API to time out and return an error when a long-running query is encountered) by setting the `async=False` flag when calling `/query`.

For large query results, the Query Service may deposit results in S3 instead of returning them verbatim in the response body. In this case, the client will receive a `302 Found` response status code sending them to the response data location. In this mode, response data are confidential to the caller, and remain accessible for 7 days. The caller may turn off this functionality by setting the `async=False` flag when calling `/query`.

CHAPTER 2

Data Schema

Because there are often multiple, slightly different versions of a bundle or file, the `bundles_all_versions` and `files_all_versions` tables contain all versions of the bundles and files. There are also derived view tables `files` and `bundles`, which only contain the latest version of each bundle or file.

The metadata itself is available in `files.body` as a JSON data object. The structure of that document is dependent on the `dcp_schema_type_name` column. The schemas for each schema type can be found here. It is also possible to pull the full JSON document for a file of a particular schema type (or the `aggregate_metadata` field for a bundle to see the combined metadata of multiple files) and explore it in your text editor to better understand what data it contains and how it is formatted. For example, to get the JSON associated with a cell line

```
SELECT body FROM files WHERE dcp_schema_type='cell_line';
```

or

```
SELECT body FROM cell_line;
```

To simplify queries, the view tables for each schema type contain only the most recent version of each metadata file as the schema of the metadata may change between versions.

Possible schema_types include `cell_line`, `cell_suspension`, `differentiation_protocol`, `dissociation_protocol`, `donor_organism`, `ipsc_induction_protocol`, `library_preparation_protocol`, `links`, `organoid`, `process`, `project`, `sequence_file`, `sequencing_protocol`, `specimen_from_organism`, `supplementary_file`, `analysis_file`, `analysis_process`, `analysis_protocol`, `collection_protocol`, `enrichment_protocol`, `biomaterial`, `file`, `image_file`, `imaged_specimen`, `imaging_preparation_protocol`, `imaging_protocol`.

CHAPTER 3

Example Queries

Get a list of all of the tables:

```
SELECT table_name
FROM information_schema.tables
WHERE table_type = 'BASE TABLE'
 AND table_schema NOT IN ('pg_catalog', 'information_schema')
 AND table_schema = 'public';
```

Get total number of bundles:

```
SELECT count(*) from bundles;
```

Get all data for 10 bundles:

```
SELECT * FROM bundles LIMIT 10;
```

Select the data for a particular bundle:

```
SELECT * FROM bundles WHERE uuid='cf48f5bc-7f20-4aa8-a0b6-6f889466546d'
```

If you query on `uuid` the database may return multiple versions of the bundle; to only get one, use the `fqid` (which is the `version` concatenated to the `uuid`)

3.1 Querying the metadata body

The metadata itself is stored in the jsonb format. For help with the syntax of querying jsonb check out this cheat sheet.

Get a list of submissions for a particular submitter:

```
SELECT p.uuid, p.body->'project_core'->>'project_title' AS title
FROM project AS p
WHERE p.body @> '{"contributors": [{"contact_name": "Aviv,,Regev"}]}'
   OR p.body @> '{"contributors": [{"contact_name": "Sarah,,Teichmann"}]}';
```

3.2 Querying the experimental graph

If you have a FASTQ file and you are curious about its inputs, you can use a custom SQL function `get_all_parents` to get all of the processes that led to its creation. For example:

If your sequence file has a uuid of `8c823b32-42dc-4163-aa96-168dce981ee5` you can run:

```
SELECT * FROM process_file_join_table
WHERE file_uuid = '8c823b32-42dc-4163-aa96-168dce981ee5';
```

This should return a list of all the processes the file is associated with. Find a row that says `OUTPUT_ENTITY` and copy the `process_uuid`.

Then run the following to get a list of all of its parent processes:

```
SELECT * FROM  get_all_parents('4028ba54-e09f-4c16-9b4e-4f0781e80c46')
```

Things to note about the query above:

- Direct parents are processes whose output becomes the input for the current process
- `get_all_parents` grabs the processes whose output was the input for the current process and then gets the processes whose output was the input for the parent process and then goes all the way up to the initial file (typically a `donor_organism` file)
- You can also use the `get_all_children` function to go the opposite way
- This allows us to represent graphical data in a relational database

`donor_organism` files are typically at the top of the graph. If you have a `file_uuid` for a `donor_organism` and you'd like to get all of the `sequence_files` that came from that donor, you can run the following substituting in the `file_uuid` of your `donor_organism`.

```
SELECT *
FROM (SELECT file_uuid
      FROM process_file_join_table
      WHERE process_uuid IN (SELECT get_all_children(process_uuid)
                             FROM process_file_join_table
                             WHERE file_uuid = '79a7c087-886b-47a7-8044-76a227d963ba
↪')) AS temp_table
       JOIN files ON temp_table.file_uuid = files.uuid
WHERE files.dcp_schema_type_name = 'sequence_file';
```

Further notes on the above query:

- To get all file types, leave off the final `WHERE` clause
- To retrieve files higher in the graph, replace `get_all_children` with `get_all_parents`

If you are trying to get all of the files in a graph, it is necessary to specifically retrieve:

- Any input files and protocol files for a process when running `get_all_children`
- Any output files and protocol files for a process when running `get_all_parents`