DMAP API

Overview

The DMAP API is a GraphQL-like API that will allow users to query all the tables in the Envirofacts database. The backend is written in Lambda functions using Chalice. It is not a true GraphQL implementation as it lacks the ability to use fragments, perform introspection, have a schema file, and a host of other functionality.

Endpoints

The production endpoint is located at https://data.epa.gov/dmapservice/query. The internal endpoint is located at https://internal.aws-prod.aws.epa.gov/api/query.

Quickstart

The basic structure of a query consists of a schema, table, and fields. All examples below are for tables and schemas which do not currently exist. They are simplified examples to clearly indicate functionality.

Example

```
query sampleQuery {
    users__contact_list {
        id
        name
        address
    }
}
```

This query will select the fields id, name, and address from the contact_list table in the users schema. It will return all the records up to the current limit of 10,000 records.

Schema and Table

A schema and table must be supplied when querying data. The format is schema**table. For example, when to query the contact_list table in the users schema, the format would be `users**contact_list`.

Table Aliases

A table can be renamed in the resulting data set with the use of the alias function.

```
query sampleQuery {
    users__contact_list (alias: contacts) {
```

```
        id
        name
        address
    }
}
```

In the example above, the contact*list data will be under the "contacts" element in the resulting data set. The original schema and table name are not referenced in the results. _The table alias is not used when exporting results in the graphql format.*

Selecting Fields

At least one field must be supplied (if not querying an aggregate column, see below) in a query. Fields are specified as a nested list within the table element. There is no delimiter between fields. In the query:

```
query sampleQuery {
    users__contact_list {
        id
        name
        address
    }
}
```

The 3 fields id, name, and address will be returned in the result set.

All fields

If you want to return all the fields in a table, use the special field `__all_columns__`. You do not need to include any other columns in the table.

```
query sampleQuery {
    users__contact_list {
        __all_columns__
    }
}
```

Field Aliases

To rename a column's name for the purposes of a query, use the alias function with the value you want the column name's output to be in the result set.

```
query sampleQuery {
    users__contact_list {
        id
        name
        address (alias: "street_address")
    }
```

```
}
```
In the example above, the address column will be returned as street_address in the JSON object array.

Field Casts

To cast a selected field to a different type when it is returned, add the cast attribute to the field in the query

```
query sampleQuery {
    users__contact_list {
        id
        name
        age (cast: "text")
    }
}
```
In this example, the age field will be returned as a text type instead of an integer.

Valid Cast Types

- date
- decimal
- float
- integer
- number
- text

Aggregations

To perform an aggregation method on a table (e.g. count, min, max), the aggregate table needs to be queried. This aggregate table can have any filter and join applied to it. As an example of querying the count on the contact_list table, the table queried would be `users__contact_list__aggregate`. Then instead of listing the fields, query the count object within the aggregate.

```
query sampleQuery {
    users__contact_list__aggregate {
        aggregate {
            count
        }
    }
}
```
Available aggregation methods

- count
    - Purpose: Returns the total count of records.

    - Format: `count`
- max
    - Purpose: Get the maximum value of a column.
    - Format: `__max__column_name`
- min
    - Purpose: Get the mininum value of a column.
    - Format: `__min__column_name`

Distinct

To add a distinct clause in the query, add an argument to the primary table that specifies `distinct: true`.

```
query sampleQuery {
    users__contact_list (distinct: true) {
        id
        name
        address (alias: "street_address")
    }
}
```

Functions

To select a function in the results, add a `parameters` parameter that contains an array of the data to pass to the function.

```
query sampleQuery {
    users__contact_list {
        id
        name
        address (alias: "street_address")
        users__name_concat (parameters: ["John", "Smith"])
    }
}
```

Standard Functions

To use a standard function in the query, add `fn__` before the function name. Currently only coalesce, coalesceText and count are supported. Pass any parameters in the parameters argument.

fn__add will add the values of any columns and numbers passed in as parameters. fn__coalesce will return the value in the database field supplied as the second parameter if the database field in the first parameter has a NULL value. fn__coalesceText will return the text supplied as the second parameter if the database field in the first parameter has a NULL value. fn__count will return the count of the database field supplied in the first parameter. fn__dateadd will add or subtract the interval provided to the date supplied in the first parameter. The second parameter is the date part to be added or subtracted. alid values for precision are: microseconds, milliseconds, second, minute, hour, day, week, month, quarter, year, decade, century, millennium. The third parameter is the

interval to be applied. fn__datetrunc will truncate a date database field supplied in the first parameter. The second parameter is the precision of the date field to be retrieved. Valid values for precision are: microseconds, milliseconds, second, minute, hour, day, week, month, quarter, year, decade, century, millennium. The third parameter is optional and is the time zone (e.g. America/New_York). fn__divide will divide the values of any columns and numbers passed in as parameters. fn__extract (fn__datepart is an alias) will return the specified part from a date database field supplied in the first parameter. The second parameter is the part of the date field to be retrieved. Valid values for the date part are: microseconds, milliseconds, second, minute, hour, day, dow (day of week), doy (day of year), week, month, quarter, year, decade, century, millennium. fn__lpad will return a padded version of the database field supplied in the first parameter. The second parameter is the length of the total number of characters to be returned. The third parameter is the character used to pad the database field. fn__ltrim will return a trimmed version of left beginning of the database field supplied in the first parameter. The trimming will occur on the leading and trailing characters. The second parameter is optional (it defaults to a blank space ' ') and is the characters that are to be removed. fn__max is an aggregate function and will return the highest value based on the criteria supplied for the database field supplied in the first parameter. fn__min is an aggregate function and will return the lowest value based on the criteria supplied for the database field supplied in the first parameter. fn__multiply will multiply the values of any columns and numbers passed in as parameters. fn__replace rill return a modified version fo the database field supplied in the first parameter. The second parameter is the pattern to search for in the field's data. The third parameter is optional (it defaults to an empty string) and is the characters that will replace the pattern. fn__rtrim will return a trimmed version of right ending of the database field supplied in the first parameter. The trimming will occur on the leading and trailing characters. The second parameter is optional (it defaults to a blank space ' ') and is the characters that are to be removed. fn__subtract will subtract the values of any columns and numbers passed in as parameters. fn__sum will return the sum of the values of the chosen column. It needs to be used in conjunction with a group by clause. fn__trim will return a trimmed version of the database field supplied in the first parameter. The trimming will occur on the leading and trailing characters. The second parameter is optional (it defaults to a blank space ' ') and is the characters that are to be removed.

```
query sampleQuery {
    users__contact_list {
        id
        name
        address (alias: "street_address")
        fn__coalesce (parameters: ["name", "address"], alias:
"coalesce_result")
    }
}
```

Sum Function

The fn__sum function returns the sum of the values of the chosen column. As it is an aggregation function, if there are any other columns being selected, they need to be in a groupBy clause.needs to be used in conjunction with the groupBy clause.

```
query sampleQuery {
```

```
    employees__financials_list (groupBy: ["city", state"]) {
        city
        state
        fn__sum (parameters: ["salary"], alias: "sum_salaries")
    }
}
```

This query will return the sum of salaries with it being grouped by employee cities and states.

Selecting from a Function

To select results from a function, add a `parameters` parameter to the function and query the function as if it was a table.

```
query sampleQuery {
    users__combine_user_orders (parameters: ["123ABC456"]) {
        __all_columns__
    }
}
```

Case Statements

To enable if/else functionality when selecting fields, you can use a case statement in the query.

```
query sampleQuery {
    users__contact_list {
        name
        street_address
        city
        state_abbr (case: {if: {equals: "VA", then: "Virginia"}, elseif:
{equals: "MD", then: "Maryland"}, elseif: {equals: "DC", then: "District
of Columbia"}, else: "Other State"})
    }
}
```

In the query above, when the results are returned, if the state_abbr field is "VA", then "Virginia" will be returned in the record. Likewise if the state_abbr field is "MD" or "DC", it will return "Maryland" or "District of Columbia", respectively. If the state_abbr field is any other value, "Other State" will be returned.

If you need to create a case statement that that will be based on multiple fields, you can create a query that uses the "case" keyword.

```
query sampleQuery {
    users__contact_list {
        name
        street_address
        city
```

```
        state_abbr (case: {if: {equals: "VA", then: "Virginia"}, elseif:
{equals: "MD", then: "Maryland"}, elseif: {equals: "DC", then: "District
of Columbia"}, else: "Other State"})
    }
}
query sampleQuery {
employees__financials_list {
city
state
case (if: {fn__sum (parameters: ["salary"], greaterThanEqual: 1000000},
then: "Too much"}, elseif: {fn\_\_sum (parameters: ["salary"], lessThan:
1000000}, then: "Just right"}, else: "Not supplied", alias:
"sum_salaries")
}
}
```

Filtering

Any table can have filters applied to it. When creating the query, put a parentheses after the table with the filter criteria using the `where` operator.

```
query sampleQuery {
users\_\_contact_list (where: {state_abbr: {in: ["VA", "MD"]}}) {
id
name
street_address
city
state_abbr
}
}
```

In this query, only contacts whose state abbreviation is in "VA" or "MD" will be returned.

Multiple filters can be applied to a table, by default, multiple filters are applied using "AND" logic.

If you need to specify a filter on a separate table (for instance if you are using an "OR" clause on separate filter conditions) then qualify the column name with the schema and table in the `schema__table__column` format (each separated by two underscores).

```
query sampleQuery {
```

```
users\_\_contact_list (where: {state_abbr: {in: ["PA", "IL"]}, city:
{equals: "Springfield"}}) {
id
name
street_address
city
state_abbr
}
}
```

Here all users who reside in the city of Springfield and who live in Pennsylvania or Illinois will be returned.

Multiple filter conditions can also be applied with "OR" logic.

```
query sampleQuery {
users\_\_contact_list (where: {state_abbr: {equals: "CT"}, or: {city:
{equals: "New York"}}}) {
id
name
street_address
city
state_abbr
}
}
```

This query will return all users who live in Connecticut or their city of residence is New York.

Filter conditions can also be nested (the equivalent of parentheses in order of operators).

```
query sampleQuery {
users\_\_contact_list (where: {state_abbr: {equals: "CT"}, or: {city:
{equals: "New York"}, and: {street_address: {like: "%Avenue of the
Americas"}}}}) {
id
name
street_address
city
state_abbr
}
```

```
}
```

This query will get users who reside in Connecticut or they live in New York and their street address is on "Avenue of the Americas". (Equivalent to: (state_abbr=CT OR (city="New York" AND street_address LIKE "%Avenue of the Americas"))).

Available Filter Operators

When working with string values, most operators are case-insensitive. If applicable, there will be a case-sensitive version availabe in the format `{operator}Sensitive`.

- beginsWith
  - Purpose: Finds values that will start with the supplied value.
  - Example: `street_address: {beginsWith: "100"}`
  - Expected Result: All users with street addresses that start with 100 will be returned.
  - beginsWithSensitive is available for case-sensitive queries.
- between
  - Purpose: Finds values that are between two values, inclusive of the lower and upper values.
  - Example: `age: {between: {lower: 20, upper: 30}}`
  - Expected Result: All users who have an age greater than or equal to 20 and less than or equal to 30 will be returned.
  - betweenSensitive is available for case-sensitive queries.
- contains
  - Purpose: Finds records that have the supplied value anywhere in the field.
  - Example: `street_address: {contains: "Main"}`
  - Expected Result: All users who have an address that contains "Main" will be returned.
- endsWith
  - Purpose: Finds values that will end with the supplied value.
  - Example: `street_address: {endsWith: "St."}`
  - Expected Result: All users with street addresses that end with St. will be returned.
  - endsWithSensitive is available for case-sensitive queries.
- equals
  - Purpose: Finds values that will equal the supplied value.
  - Example: `state_abbr: {equals: "CA"}`
  - Expected Result: All users in California will be returned.
  - equalsSensitive is available for case-sensitive queries.
- excludes
  - Purpose: Finds values that will not be like the supplied value anywhere in the field (the opposite of contains).
  - Example: `street_address: {excludes: "Main"}`
  - Expected Result: All users whose street address does not include Main will be returned.
  - excludesSensitive is available for case-sensitive queries.
- greaterThan
  - Purpose: Finds values that are greater than the supplied value.

    - Example: `age: {greaterThan: 30}`
    - Expected Result: All users with an age of 31 or higher will be returned.
    - greaterThanSensitive is available for case-sensitive queries.
- greaterThanEqual
    - Purpose: Finds values that are greater than or equal to the supplied value.
    - Example: `age: {greaterThanEqual: 30}`
    - Expected Result: All users with an age of 30 or higher will be returned.
    - greaterThanEqualSensitive is available for case-sensitive queries.
- in
    - Purpose: Finds values that are in the supplied list.
    - Example: `state_abbr: {in: ["MS", "AL", "LA"]}`
    - Expected Result: All users in the states of Mississippi, Alabama, and Lousiana will be returned.
    - inSensitive is available for case-sensitive queries.
- inSubQuery
    - Purpose: Finds values that are in a subquery in the query.
    - Example: `state_abbr: {inSubQuery: insubquery: "state_query"}}`
    - Expected Result: All users whose state is supplied in the subquery aliased as "state_query" will be returned.
    - inSensitive is available for case-sensitive queries.
- lessThan
    - Purpose: Finds values that are less than the supplied value.
    - Example: `age: {lessThan: 30}`
    - Expected Result: All users with an age of 29 or lower will be returned.
    - lessThanSensitive is available for case-sensitive queries.
- lessThanEqual
    - Purpose: Finds values that are less than or equal to the supplied value.
    - Example: `age: {lessThanEqual: 30}`
    - Expected Result: All users with an age of 30 or lower will be returned.
    - lessThanEqualSensitive is available for case-sensitive queries.
- like
    - Purpose: Finds values that are similar to the supplied value. Uses the % character as a wildcard.
    - Example: `street_address: {like: "%Main St."}`
    - Expected Result: All users with an any address that begins with Main St. will be returned.
    - likeSensitive is available for case-sensitive queries.
- notBeginsWith
    - Purpose: Finds values that will not start with the supplied value.
    - Example: `street_address: {notBeginsWith: "100"}`
    - Expected Result: All users with street addresses that do not start with 100 will be returned.
    - notBeginsWithSensitive is available for case-sensitive queries.
- notEndsWith
    - Purpose: Finds values that will not end with the supplied value.
    - Example: `street_address: {notEndsWith: "St."}`

    - Expected Result: All users with street addresses that do not end with St. will be returned.
    - notEndsWithSensitive is available for case-sensitive queries.
- notEquals
    - Purpose: Finds values that are not equal to the supplied value.
    - Example: `state_abbr: {notEquals: "FL"}`
    - Expected Result: All users who do not reside in Florida will be returned.
    - notEqualsSensitive is available for case-sensitive queries.
- notIn
    - Purpose: Finds values that are not in the supplied list.
    - Example: `state_abbr: {in: ["TX", "OK", "KS"]}`
    - Expected Result: All users who are not in the states of Texas, Oklahoma, and Kansas will be returned.
    - notInSensitive is available for case-sensitive queries.
- notLike
    - Purpose: Finds values that are not similar to the supplied value. Uses the % character as a wildcard.
    - Example: `street_address: {notLike: "%Main St."}`
    - Expected Result: All users with an any address that do not begin with Main St. will be returned.
    - likeSensitive is available for case-sensitive queries.
- null
    - Purpose: Finds values that are either null or not null.
    - Example: `state_abbr: {null: true}`
    - Expected Result: All users who do not have a value for state will be returned.
    - Example: `state_abbr: {null: false}`
    - Expected Result: All users who do have a value for state will be returned.
- regex
    - Purpose: Finds values that match the supplied regular expression.
    - Example: `street_address: {regex: "(main){1,3}"}`
    - Expected Result: All users who have a street address that contains main at least once and at most three times.
    - regexSensitive is available for case-sensitive queries.

Casting fields in filters

When needing to cast a field to a different type to work with a filter, use the cast attribute in the filter.

```
query sampleQuery {
users\_\_contact_list (where: {street_number: {like: "%10%", cast:
"text"}}) {
id
name
```

```
address (alias: "street_address")
}
}
```

In the example above, the results will be filtered by the street number. Since the street number is stored as an integer, it is cast to a text field so the like operator can be used.

Valid Field Cast Types

- date
- decimal
- float
- integer
- number
- text

Using functions in a filter

You can use the LPad, Replace, and Trim (along with LTrim and RTrim) functions in the filter. To do so, add the function as an attribute for the field in the where clause.

- lpad - Left-pads a string with a specified character. Example: `(where: {age: {like: "%1", lpad: [3, "0"}})`. The first parameter is the length, the second is the character to pad with, the default pad character is a single space (" ").
- replace - Replaces the pattern in the database field values with the replacement text. Example: `(where: {email: {like: "@gmail.com", replace: ["_", "."]}}). The first parameter is the pattern to search for, the second parameter is the text to replace the pattern with, the default replacement text is an empty string.
- trim - Trim the string. Example: `(where: {age: {like: "%1", trim: "123"}})`. The parameter is the characters to trim, ihe default trim is a single space (" ").
- ltrim and rtrim - Trim the left or right of the string, similar to trim.

Dates

You can filter by dates and datetimes for fields that are of a date data type. For dates, the value must be in the format `YYYY-MM-DD`. For datetimes, the value must be in the format `YYYY-MM-DD HH:MM:SS`. Datetimes use the 24 hour format.

```
query sampleQuery {
users\_\_contact_list (where: {start_date: {greaterThanEqual: "2021-05-
21"}}) {
id
name
address (alias: "street_address")
```

```
}
}
```

In the example above, the results will be filtered to get the users who started on or after May 21, 2021.

```
query sampleQuery {
users\_\_contact_list (where: {start_date: {greaterThanEqual: "2021-06-13
15:30:00"}}) {
id
name
address (alias: "street_address")
}
}
```

In the example above, the results will be filtered to get the users who started on or after June 13, 2021 at 3:30 PM.

Joins

Tables can be joined to each other in queries. A joined table can still have filters, orderBy, and groupBy clauses applied to it.

Tables can be joined using the equals, notEquals, equalsInsensitive, and notEqualsInsensitive operators. When using the equals or notEquals operators, the join will be equalivalent to equalsSensitive and notEqualsSensitive, respectively.

```
query sampleQuery {
users__contact_list (where: {state_abbr: {in: ["VA", "MD"]}}) {
id
name
street_address
city
state_abbr
users__orders (left_outer_join: {users__contact_list__id: {equals:
user_id}}) {
order_id
order_name
order_date
}
}
```

```
}
```

This query will be run against the contact_list and orders table in the users schema. A left outer join will be performed on the contact_list table's id column and the orders table's user_id column.

Join Operator Types

- `cross_join`: Cross Join
- `full_outer_join`: Full Outer Join
- `hash_join`: Hash Join
- `inner_join`: Inner Join
- `left_join`: Left Join
- `left_outer_join`: Left Outer Join
- `right_join`: Right Join
- `right_outer_join`: Right Outer Join
- `outer_join`: Outer Join

Functions in Joins

Joins support the use of functions when joining tables. Trim (along with LTrim and RTrim), LPad, Replace, and Cast are supported. Here is an example of a join using lpad:

```
query joinQuery {
users__contact_list {
name
street_address
form_type_ind
users__orders (left_outer_join: {users__contact_list__id: {equals:
{user_id: {lpad: [10, '0']}}, lpad: [10, '0']}}) {
order_id
order_name
order_date
}
}
}
```

Subqueries

The API supports a limited version of subqueries. A subquery can be created similar to a join, only using the `subquery` parameters. A subquery can still have filters, orderBy, and groupBy clauses applied to it.

```
query sampleQuery {
users__contact_list (where: {state_abbr: {in: ["VA", "MD"]}}) {
id
name
street_address
city
state_abbr
users__orders (subquery: {alias: "orders"}) {
order_id
order_name
order_date
}
}
}
```

This query will be run against the contact_list and orders table in the users schema. A subquery will be performed on the orders table as it pulls in the order_id, order_name, and order_date columns.

If the subquery is to be used with the inSubquery filter, then the alias parameter must be supplied!

Pagination

To paginate through results, the `limit` and `offset` parameters can be supplied.

```
query sampleQuery {
users\_\_contact_list (limit: 10, offset: 20) {
id
name
street_address
city
state_abbr
}
}
```

In this query, a maximum of 10 records will be returned. The first 20 matching records in the contact_list table will be skipped and the next 10 records will be returned.

The API will return a maximum of 10,000 records.

Ordering Results

Results can be ordered through the orderBy query parameter.

```
query sampleQuery {
users\_\_contact_list (orderBy: {city: "asc", state_abbr: "desc"}) {
id
name
street_address
city
state_abbr
}
}
```

The results in this example will be ordered by first the city field ascending and then by the state_abbr field descending. `"asc"`, `"ascNullsFirst"`, `"ascNullsLast"`,`"desc"`, `"descNullsFirst"`, and `"descNullsLast"` are the only valid values for orderBy.

Ordering Joined Columns

When joining tables and wanting to order the results by columns in a joined table, specify all the joins in the primary table. For the columns in the joined table, use the format `schema__table__column` to specify the column that should be joined. (The schema, table, and column are separated by double underscores `__`.)

```
query sampleQuery {
users__contact_list (where: {state_abbr: {in: ["VA", "MD"]}}, orderBy:
{name: "asc", users__orders__order_date: "desc"}) {
id
name
street_address
city
state_abbr
users__orders (left_outer_join: {users__contact_list__id: {equals:
user_id}}) {
order_id
order_name
order_date
}
}
}
```

In this query, the results will be ordered first in ascending order by the name column in the users.contact_list table. Then the results will be ordered by the order_date column in the users.orders table in descending order.

Grouping Results

```
query sampleQuery {
users\_\_contact_list (groupBy: ["city", "state_abbr"]) {
id
name
street_address
city
state_abbr
}
}
```

The results in this example will be grouped by the city and state_abbr fields.

Variables

As with normal GraphQL queries, variables can be substituted for values in query parameters, for example in the limit field or the filter values. The allowed types of values are String, Int, Float, and OrderBy. The String, Int, and Float values can also be stored in arrays.

Example Query with limit, offset, and order by variables

Query

```
query query($limit: Int!, $offset: Int!, $orderBy: OrderBy!]) {
users__contact_list (limit: $limit, offset: $offset, where: {state_abbr:
{in: ["VA", "MD"]}}, order_by: $orderBy) {
__all_columns\_\_
}
}
```

Variables

```
{"limit": 10, "offset": 30, "orderBy": {"city": "asc", "state_abbr":
"desc"}}
```

Here the limit, offset, and orderBy variables will be substituted in the query when it is executed against the data source. The $limit, $offset variables are required integers. The $orderBy variable is a required variable in the order by clause structure.

Example Query with where clause variables

Query

```
query query($state: [String]) {
users__contact_list (state_abbr: {in: $state}}) {
__all_columns\_\_
}
}
```

Variables

```
{"state": ["ND", "SD"]}
```

Here the $state variable is an array of strings. The values of "ND" and "SD" will be provided to the query when it runs against the database.

Multiple queries

Multiple queries can be sent in the same request.

```
query sampleQuery {
users__contact_list {
id
name
street_address
city
state_abbr
}
users__contact_list\_\_aggregate {
aggregate {
count
}
}
}
```

They will be run sequentially. This increases the chance of a timeout on the request with no results being returned.

Returned data

Data is returned from the service in a nested JSON object. The primary element is called data. Each table's results will be returned as a property within the data element. For example:

```
{
"data": {
"users__contact_list": [
{ "id": 123, "name": "John Doe", "address": "123 Main Street" },
{ "id": 124, "name": "Jane Doe", "address": "456 Broad Street" },
.
.
.
],
"users__contact_list\_\_aggregate": [
{ "count": 456 }
]
}
}
```

Export formats

Data can also be returned in the following formats: CSV, Excel, JSON, JSONP, XML, HTML, PDF, and GraphQL. To export a query's results to a different format, append the format to the end of the URL. If using JSONP, add the callback to wrap the result in with the "callback" query parameter.

CSV: /query/csv Excel: /query/excel XML: /query/xml JSON: /query/json JSONP: /query/jsonp?callback=callback-value HTML: /query/html PDF: /query/pdf GraphQL: /query/graphql

The GraphQL export format is similar to the regular format (/query). This format is provided to have greater compatibility with some GraphQL clients.