How to access processed data

The `ArchiveQuery` allows you to search for entries and access their parsed and processed *archive* data

at the same time. Furthermore, all data is accessible through a convenient Python interface

based on the schema rather than plain JSON. See also this guide on using NOMAD's Python schemas (../plugins/schema_packages.md#use-python-schemas-to-work-with-data)

to work with processed data.

As a requirement, you have to install the `nomad-lab` Python package. Follow the How to install nomad-lab (pythonlib.md) guide.

Getting started

To define a query, one can, for example, write

```python
from nomad.client.archive import ArchiveQuery
query = ArchiveQuery(query={}, required='*', page_size=10, results_max=10000)
```

Although the above query object has an empty query.

The query object is constructed only.

To access the desired data, users need to perform two operations manually.

Two interfaces that can be used in different environments are provided.

Synchronous Interface

Fetch

The fetch process is carried out **synchronously**. Users can call the following to fetch up to `results_max` entries.

```python
```

number_of_entries = query.fetch(1000) # fetch 1000 entries

number_of_entries = query.fetch() # fetch at most results_max entries

```

An indicative number `n` can be provided `fetch(n)` to fetch `n` entries at once.

The fetch process may submit multiple requests to the server, each request asks for `page_size` entries.

The number of qualified entries will be returned.

Meanwhile, the qualified entry list would be populated with their IDs.

To check all qualified upload IDs, one can call `entry_list()` method to

return the full list.

```python

print(query.entry_list())

```

If applicable, it is possible to fetch a large number of entries first and then perform a second fetch by using some

entry ID in the first fetch result as the `after` argument so that some middle segment can be downloaded.

Download

After fetching the qualified entries, the desired data can be downloaded **asynchronously**. One can call

```python

results = query.download(1000) # download 1000 entries

results = query.download()  # download all fetched entries if fetched otherwise fetch and download up to `results_max` entries

```

to download up to `results_max` entries. The downloaded results are returned as a list.
```

Alternatively, it is possible to

just download a portion of previously fetched entries at a single time. For example,

```python
previously fetched for example 1000 entries
but only download the first 100 (approx.) entries
results = query.download(100)
```

The same `download(n)` method can be called repeatedly. If there are no sufficient entries, new entries will be

automatically fetched. If there are no more entries, the returned result list is empty. For example,

```python
total_results = []
while True:
    result = query.download(100)
    if len(result) == 0:
        break
    total_results.extend(result)
```

There is no retry mechanism in the download process.

If any entries fail to be downloaded due to server error, it is kept in the list otherwise removed.

Pandas Dataframe

You can also convert the downloaded results to pandas dataframe directly by calling `entries_to_dataframe` method

on the `query` object. In order to filter the final dataframe to contain only specific

keys/column_names, you can

use the option `keys_to_filter` with a list of relevant keys. For example:

```python

results = query.entries_to_dataframe(keys_to_filter=[])

```

The option `from_query` can be used to control the formatting of the dataframe(s). By setting this option to `False`,

all entries with their entire contents are flattened and returned in one single (and potentially huge) dataframe,

and by setting it to `True`, it returns a python dictionary with each key denoting a separate distinct nested path

in the `required` and each value specifying the corresponding dataframe.

Asynchronous Interface

Some applications, such as Jupyter Notebook, may run a global/top level event loop. To query data in those environments,

one can use the asynchronous interface.

```python

number_of_entries = await query.async_fetch()  # indicative number n applies: async_fetch(n)

results = await query.async_download()  # indicative number n applies: async_download(n)

```

Alternatively, if one wants to use the asynchronous interface, it is necessary to patch the global event loop to allow

nested loops.

To do so, one can add the following at the beginning of the notebook.

```python
import nest_asyncio
nest_asyncio.apply()
```

A Complete Rundown

Here we show a valid query and acquire data from server.

We first define the desired query and construct the object.

We limit the maximum number of entries to be 10000 and 10 entries per page.

```python
from nomad.client.archive import ArchiveQuery
required = {
    'workflow': {
        'calculation_result_ref': {
            'energy': '*',
            'system_ref': {
                'chemical_composition_reduced': '*'
            }
        }
    }
}
query = {
    'results.method.simulation.program_name': 'VASP',
    'results.material.elements': ['Ti']
}
query = ArchiveQuery(query=query, required=required, page_size=10, results_max=10000)
```

```

Let's fetch some entries.

```python
query.fetch(10)
print(query.entry_list())
```

If we print the entry list, it would be

```text
[('---CU_ejqV7yjEFteUAH0rG0SKlS', 'aimE2ajMQnOKruRbQjzpCA'),
('---Dz9vL-eyErWEk7-1tX4zLVmwo', 'Vyb6k1OTRSuyXfvsRk4CPQ'),
('---pRcX7NG_XDx_4ufUaeEnZnmrO', 'lL_YBCD8TSyLlsqzIcBgYw'),
('--0RSTtl4mvX3Nd0JjhL_V1YV1ip', 'tF8R9nmZTyyfnv2zWADI0A'),
('--0SDuSOM_gpweM3PDb0WOFgYDyv', 'mLO6o1GBShWrtXfoSJHgfw'),
('--0jLz1eNRwtR_oRxPpgVC9U437y', 'HVheHWfxTpe28HhbHpcO1A'),
('--1cY4hzXaXdThxsw7saN3nd3xyt', 'h79v1yw_Qf-kOVTa0WYfdg'),
('--2nakQLLxyI_vsEOIbwzHMgWWPQ', 'iIGUoyaiT5i4b4UynPlnSQ'),
('--3-SQGOswGzwaEo5QiQNCcZQhi8', '1lQ90kNaSWyJXBQ_kK91Tg'),
('--3Km0GSVTHRkNCJHHjQdHqxwVfR', 'dHAJP-NvQw22FoBeDXiAlg')]
```

Each is a tuple of two strings. The first is the entry ID while the second is the upload ID.

Now data can be downloaded.

```python
result = query.download(8)
print(f'Downloaded {len(result)} entries.')  # Downloaded 100 entries.
```

The first eight entries will be downloaded.
```

If one prints the list again, only the last two are present.

```python
print(query.entry_list())
```

```text
[('--3-SQGOswGzwaEo5QiQNCcZQhi8', '1lQ90kNaSWyJXBQ_kK91Tg'),
('--3Km0GSVTHRkNCJHHjQdHqxwVfR', 'dHAJP-NvQw22FoBeDXiAIg')]
```

It is possible to download more data.

We perform one more download call to illustrate that the fetch process will be automatically triggered.

```python
result = query.download(5)
print(f'Downloaded {len(result)} entries.')  # Downloaded 200 entries.
```

In the above, we request five entries.

However, the list contains only two entries, the fetch process will be called to fetch extra three entries from server.

But since the page size is 10, the server will return 10 entries.

You will see the following message in the terminal.

```text
Fetching remote uploads...
10 entries are qualified and added to the download list.
Downloading 5 entries...  [##############################################]
100%
```

Now that we have downloaded a few entries, we can convert them into pandas dataframe.

```python
dataframes = query.entries_to_dataframe(keys_to_filter=['workflow2.results.calculation_result_ref'])
print(dataframes)
```

By setting `keys_to_filter` to `['workflow2.results.calculation_result_ref']`, we create a dataframe representing the
content that exists in the response tree under the section `calculation_result_ref`. Below, you can see the final
dataframe printed in the Python console. You can also try setting this option to an empty list
(or simply removing the option) to see the results containing the entire response tree in dataframe format. Furthermore,
since we have converted our data into a pandas dataframe, we can proceed to export CSV files, obtain statistics,
create plots, and more.

```text
   energy.fermi  ...  system_ref.chemical_composition_reduced
0           NaN  ...                                    O3Ti
1           NaN  ...                                   LiO3Ti
2           NaN  ...                                    O3Ti
3           NaN  ...                                    O3Ti
4           NaN  ...                                   LiO3Ti
5           NaN  ...                                    O3Ti
```

```
6          NaN  ...                    Cu44OTi
7  -1.602177e-18  ...                    O51Ti26
8  -1.602177e-18  ...                    O51Ti26
9          NaN  ...                    O3Ti
10 -1.602177e-18  ...                    O51Ti26
11 -1.602177e-18  ...                    O51Ti26
12 -1.602177e-18  ...                    O51Ti26
13         NaN  ...                    O3Ti
14         NaN  ...                    KO3Ti
15         NaN  ...                    KO3Ti
16         NaN  ...                    O3Ti
17         NaN  ...                    AlO3Ti
18         NaN  ...                    O3TiZn
[19 rows x 7 columns]
```

Argument List

The following arguments are acceptable for `ArchiveQuery`.

- `owner` : `str` The scope of data to access. Default: `'visible'`
- `query` : `dict` The API query. There are no validations of any means carried out by the class, users shall make sure
  the provided query is valid. Otherwise, server would return error messages.
- `required` : `dict` The required quantities.
- `url` : `str` The database url. It can be the one of your local database. The official NOMAD database is used be
  default if no valid one defined. Default: `http://nomad-lab.eu/prod/v1/api`
- `after` : `str` It can be understood that the data is stored in a sequential list. Each

upload has a unique ID,

if `after` is not provided, the query always starts from the first upload. One can choose to query the uploads in the

middle of storage by assigning a proper value of `after`.

- `results_max` : `int` Determine how many entries to download. Note each upload may have multiple entries.
- `page_size` : `int` Page size.
- `username` : `str` Username for authentication.
- `password` : `str` Password for authentication.
- `retry` : `int` In the case of server errors, the fetch process is automatically retried every `sleep_time` seconds.

This argument limits the maximum times of retry.

- `sleep_time` : `float` The interval of fetch retry.

The complete example

!!! warning "Attention"

This examples uses the new `workflow2` workflow system. This is still under development

and this example might not yet produce results on the public nomad data.

```python
--8 < -- "examples/archive/archive_query.py"
```