# DataScope APIs


Haobin Cui

Durham University Business School

Durham University

A thesis submitted for the degree of

*Doctor of Philosophy*

2024

# Contents



**References** 25

# List of Figures

# Listings

# List of Tables

# Chapter 1

# Python SDK for DataScope API

## 1.1 Introduction

Data analysis plays an important role in quantitative finance analysis. Refinitiv DataScope is one of the largest data provider in the world. Users can access to highly frequent data though their database. But there are some disadvantages using DataScope. First of all, the data extraction process is complex by using DataScope's website. Also, they only provide a C# Software Development Kit (SDK) for their database's APIs. On the other hand, when users try to extract large amount of data at once, it will take a long time waiting for the response from DataScope's server. Finally, the extracted data is usually packaged in one .csv file, users may have problems opening the data file if the data size is large. To solve these problems, I have developed a python SDK to accessing DataScope's API, and use multi-thread to reduce the time for waiting the response. In this chapter, I will introduce this SDK and give some examples on how to use and how to develop using this SDK.

This chapter is structured as follows. In section 1.2, I will briefly talk about the process of getting data from DataScope. Then, in section 1.3, I will introduce the construct of this SDK and the implement of the data extraction process by this SDK. Examples on extracting some data will be give in section 1.4. Finally, I will illustrate how to use the SDK to add other features of DataScope in section 1.5, and give an example in section 1.6.

## 1.2 Process of How to Get the Data

DataScope has provide a series of REST APIs to access their database. Users can access the data using C#, .NET SDK and HTTP Urls. Sending HTTP requests and receiving responses though Python can be achieved by the builtin package "request". Therefore, in this SDK, communications between DataScope's server and the client are made by the HTTP Urls.

Due to the large size of data, at least two communications are needed for one data extraction. To start with, on the first communication, users need to tell DataScope what kind of data users need. Then, DataScope can prepare the selected data for the user. Then, on the second communication, user can access to the target data if it has been prepared by the server. This process is shown the figure 1.1.

In the process, three connections have been created for one extraction request. At the start of the first connection, a token for authorization purpose is created,according to DataScope's APIs guide, the token is valid for 24 hours and no usage limit during that time. After that, the client will send the extraction request to DataScope's server. The server will start preparing data one the re-

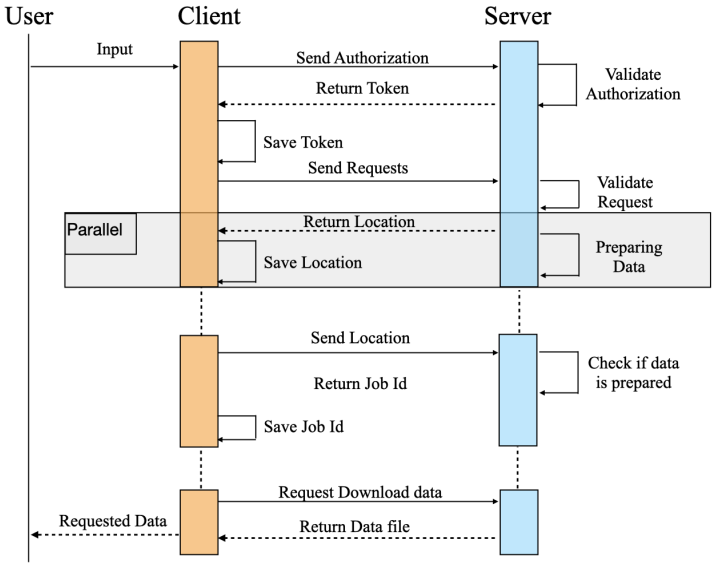


Figure 1.1: Sequence Diagram for extraction

quest is received. Also, the server will send a “location url” back with status code ”202 (accept)” and stop this connection at the same time. Users can use the location url to check whether the data is prepared or not by sending a “get” request using the “location url”. If the data is available, DataScope’s server will send the “job id” back to the client with status code “200 (OK)”. Finally, users can download the data file by the job id. Specially, if the data size is relatively small, the server will send the “job id” back directly instead of the “location url” at the first connection. As a result, a special handle is made in this SDK, which will be discussed in the next section.

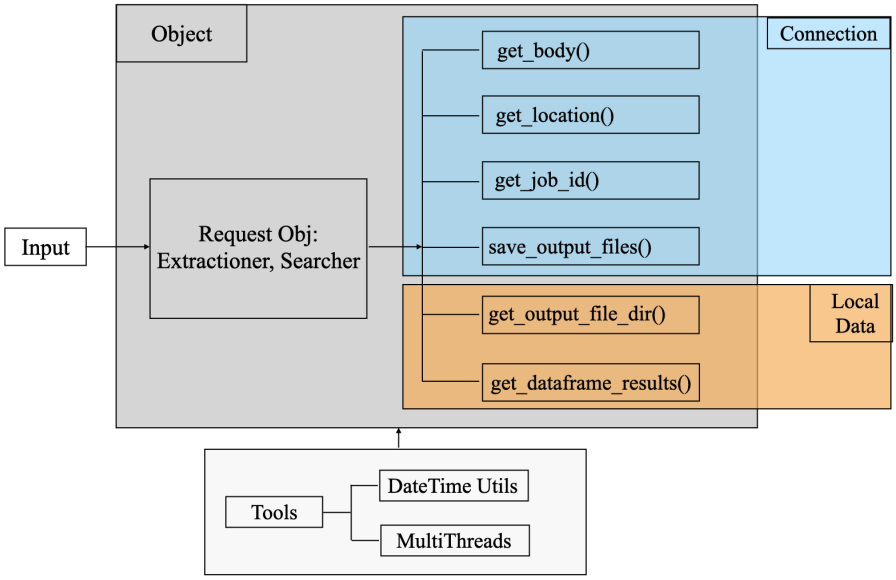


Figure 1.2: Framework of the SDK

## 1.3 Framework for the SDK

In this section, the framework of the SDK will be discussed and the corresponding codes will be shown. This SDK is an implementation for the process described in the figure 1.1 by using Object-Oriented Programming (OOP). The SDK can be separated into three main parts: Input, Request Object, and Utility tools. Figure 1.2 is a sketch map of the construct of this SDK.

First of all, Input contains the information to describe the kind of data and the range of data that the user want. For example, users can ask for market depth data or intra-day price data. Expect the identifier id which is a strings, other input information is either an enumerator object or datetime object. List 1.1 shows an example of input for extracting market depth data.

```
1
    ## ids:
3   identifier = 'CLZ24'
    identifier_type = IdentifierType.Ric
```

```
5
    ## dates (%Y, %M, %D, %h, %s, %ms):
7   query_start_date = datetime(2022, 10, 2, 0, 0, 0)
    query_end_date = datetime(2022, 10, 4, 0, 0, 0)
9
    ## content filed names:
11  bid_price = MarketDepthContentFieldNames.BidPrice
    bid_size = MarketDepthContentFieldNames.BidSize
13
    content_field_names = [bid_price, bid_size]
```

Listing 1.1: Example of Input

In the input, first of all, DataScope supports different types of identifier types, the enumerator class "IdentifierType" has listed all these types. Each kind of extraction request requires different type of content field. In the input example (list 1.1), "bid price" and "bid size" can be extracted in the market depth report. On the other hand, for a intra-day summary extractions, users can extract "close price", "open price" and etc. All these content field names is an enumerator class in file "connection.features.extraction.enums.content_field_names". List 1.2 shows the content field names for market depth extraction.

```
    class MarketDepthContentFieldNames(Enum):
2       AskPrice = 'Ask Price'
        AskSize = 'Ask Size'
4       BidPrice = 'Bid Price'
        BidSize = 'Bid Size'
6       ExchangeTime = 'Exchange Time'
        NumberOfSellers = 'Number of Sellers'
8       NumberOfBuyers = 'Number of Buyers'
```

Listing 1.2: Market Depth Extraction Content Field Names

The second part is the request object. In this SDK, all the request is deserialized as an object. For example, the market depth extraction request is deserialized as class "TickHistoryMarketDepthExtractioner". Hence, operations on the data are based on the corresponding object "extractioner". Also, all the "extractioners" are inherited from the supper class "OnDemandExtractioner". On the other hand, for search request, the parent class is "Searcher". The following codes (list 1.3) show members and members functions for the supper class "OnDemandEx-

tractioner". All these class can be found in file "connection.features".

```
    class OnDemandExtractioner(ABC):
        """
        abstract class for on demand extractioner
        """
        _IdentifierList = 'IdentifierList'
        _Condition = 'Condition'
        _UseUserPreferencesForValidationOptions = '
            UseUserPreferencesForValidationOptions'
        _ContentFieldNames = 'ContentFieldNames'
        _ExtractionRequest = 'ExtractionRequest'
        _odata_type = '@odata.type'
        _extraction_request_header = "#DataScope.Select.Api.Extractions.
            ExtractionRequests."

        def __init__(self):
            self.extraction_type = ExtractionTypes.ExtractRaw
            self.body = None
            self.job_id = None
            self.location = None
            self.output_file_path = None

        @abstractmethod
        def get_body(self) -> dict:
            raise NotImplemented

        def get_location(self, token=None) -> str:
            if self.location:
                return self.location

            body = self.body if self.body else self.get_body()
            location = post_extractions_request(self.extraction_type, body,
                token)

            self.location = location

            return location

        def get_job_id(self, token=None) -> str:
            if self.job_id:
                return self.job_id

            location = self.location if self.location else self.get_location(
                token)
            job_id = get_job_id_by_location(location, token)

            self.job_id = job_id
            return job_id

        def save_output_file(self, output_file_path: str, token=None) -> bool:
            """
            Save the .gz file to local
            :param output_file_path: Output file name need to end with .csv.gz
            :param token:
            :return: True is saved successfully
            """
            job_id = self.job_id if self.job_id else self.get_job_id(token)

            logging.info(f'Successfully get the job id, Start download the file'
                )
```

```
56              res = get_extraction_file_by_job_id(self.extraction_type, job_id,
                    output_file_path, token)

58              self.output_file_path = os.path.abspath(output_file_path)
                logging.info(f'Successfully saved file {output_file_path}')
60
                return res
62
            def get_output_file_dir(self) -> str:
64              if self.output_file_path:
                    return self.output_file_path
66              else:
                    return 'Output File Not Found'
68
            def get_dataframe_results(self, token=None) -> pd.DataFrame:
70              res = pd.read_csv(self.output_file_path if self.output_file_path
                    else self.get_output_file_dir())
                return res
```

Listing 1.3: On Demand Extractioner

In this super class (list 1.3), five methods have been implemented including one abstract method. Referring to figure 1.1, "get_location()"and "get_job_id()" will return the value of "location" and "job id" in the first and second connection with the server. "save_output_file()" will download the extracted data file and save it to the local path in the final connection. In these methods, calling "post_extractions_request()", "get_job_id_by_location()", and "get_extraction_file_by_job_id()" will send HTTP requests to DataScope's server. Also, "headers" and "bodies" of these requests will be constructed when calling these methods. These algorisms can be found in file "connection.client". "post_extractions_request()" is the first connection to the server after get the authorized token, and the location url will be received after this connection. Then, the second method "get_job_id_by_location()" will return the job id of the extraction request though the location url, each extraction request should have an unique job id. Also, a note about the data is returned with the job id. Finally, "get_extraction_file_by_job_id()" will download the extracted data file in the third connection by the unique job id. List 1.4 gives the example of the function "post_extractions_request()". The

request is sent using package “request”. If the response status code is 202, it means the server return the “location url” back. However, if the status code is 200, the “job id” is returned. It usually happens when the required data size is small, the server can finish preparation of the data in a short time. Therefore, a special handle of assembling the “job id” into the form of “location url” is made.

```
    def post_extractions_request(extraction_type: ExtractionTypes, body: dict,
        token=None):
    """
    Get the location url for the extraction request, the start point of getting
        extractions
    :return: location id or job id (if there is no output data)
    """
    url = _config.extraction_url + extraction_type.name
    headers = {
        'Authorization': 'Token ' + (token if token else _get_token()),
        'Prefer': 'respond-async',  # return the location id in the response, if
             failed, usually return job id
        'Content-Type': 'application/json; odata=minimalmetadata'
    }
    logging.debug('Start sending extraction request')
    r = requests.post(url, json=body, headers=headers)

    if r.status_code == 202:
        location = r.headers['location']
        return location

    if r.status_code == 200:
        res = r.json()
        logging.debug(f"get JobId [{res['JobId']}] with Notes [{res['Notes']}]")
        job_id = res['JobId']
        location = url + "Result" + f"(ExtractionId='{job_id}')"
        return location
    else:
        res = r.json()
        raise ValueError(f"Failed to send request for {extraction_type.name}
            with error {res['error']}")
```

Listing 1.4: HTTP Request Example

Besides, “get_body()” is the starting point for an extraction, it will return the body part in the HTTP request which contains the requirement of the data. It is the serialized dictionary of the extraction object. The body is different for each report. Therefore, implementations of this method are variance from different sub-classes. For example, the implementation for market depth extraction is listed in list 1.5.

```
    def get_body(self) -> dict:
        if self.body:
            return self.body
        body = {
            self._ExtractionRequest: {
                self._odata_type: self._extraction_request_header + self.
                    extraction_data_types.value,
                self._ContentFieldNames: [field_name.value for field_name in
                    self.market_depth_content_field_names],
                self._IdentifierList: self.identifier_list.get_dict_form('
                    Extractions.ExtractionRequests'),
                self._Condition: self.condition.dict_form,
            }
        }
        self.body = body
    return body
```

Listing 1.5: Market Depth Extraction Body

The body will be sent to DataScope's server once user call the method "post_extractions_reques Finally, "get_output_file_dir()" will return the path of the extraction file and "get_dataframe_results()" will read the extraction file and load all the values to a pandas.DataFrame for further analysis. There is no new connection created when calling these two method.

The final part is the Utility Tools, which contains some utility tool functions for calculation. Two main tools are included in this part, "datetime utilities" and "multi thread implements". "datetime utilities" contains functions used for date and time calculations, such as a function used to convert the date or datetime into a different format. "multi thread implements" is used to increase the speed when extracting large dataset.

Data extraction will spend lots of time waiting for the response from the server even for a small size of data. For example, extracting one day's one-hour summarized bid price of a stock will take about 20 seconds. However, DataScope's APIs allows to send up to 50 requests per time. A standard approach to increase the speed of I/O intensive jobs is using multiple threads. Therefore, in this SDK, there is a separation of the entire data into multiple sub-data according the query

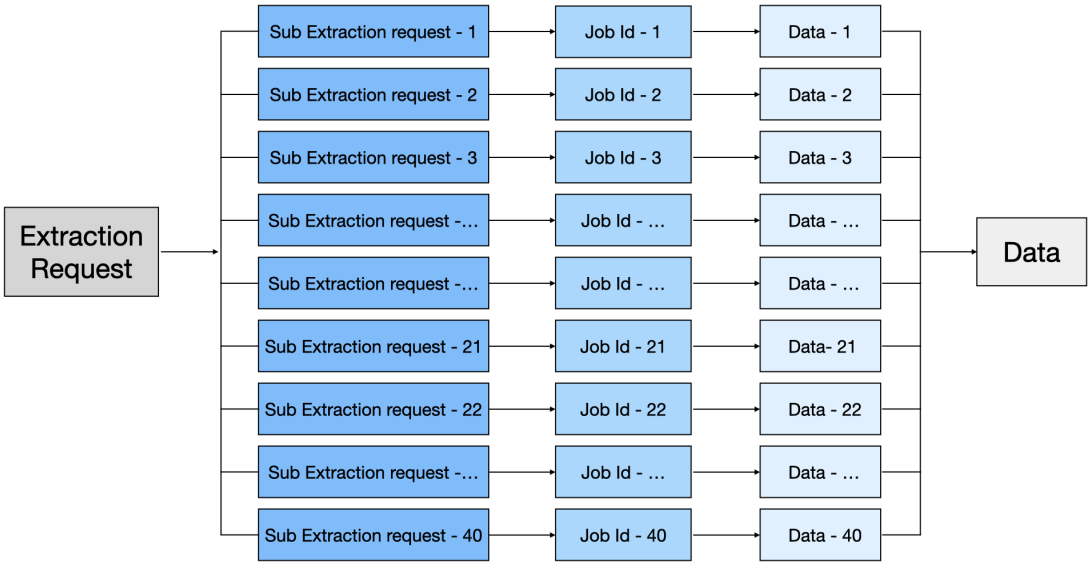


Figure 1.3: MultiThreads Implementation

time range and send them at once by implementing multi-thread. Figure 1.3 give a sketch map of the process.

The Global Interpreter Lock (GIL), a mutex which is set to protect variables, prevent deadlock and make the thread safe, python allows only one thread is executing at the same time. As a result, for CPU intensive jobs, other threads will not execute only until the current thread processed its works and the GIL released the lock. However, for I/O intensive jobs, the thread need to spend more time waiting for responses from the server rather than doing calculations. The interpreter can switch between different threads and execute codes while the current thread is wait ing for response. Hence, the performance of multi-threaded programs for I/O-bound code is not affected by the GIL. A comparison of the speed of multi-thread and non-multi-thread when extracting the same data is provide in Table 1.1.

| Number of Thread | Time Usage (second) |
|---|---|
| 1 | 487.0312 |
| 10 | 313.4916 |
| 31[1] | 139.6144 |

Table 1.1: Extracting Speed Comparison

Table 1.1 summarized the result of the time usages of extracting same market depth data for 2-level bid price of "CLZ24" from 2022/10/01 to 2022/11/01. The total size of the extracted data is 71.7 MB (compressed files) including millions of rows. The result shows that increasing the number of threads can efficiently reduce the time spending for extraction. However, for small size data, the server will spend similar time to send the response back. Hence, using multi-thread cannot increase the efficiency of data extraction. As a result, in this SDK, the minimum time unit for date time separation is set as daily.

On the other hand, the data size for market depth is usually extremely large. Not only saving the data is time consuming, but also processing the large dataset is a problem. Therefore, separating the entire dataset into different small parts can also increase the efficiency of processing the data.

The implementation of multi-threads is in "class MultiThreads", list 1.6 gives the detail of the class.

```
    class MultiThreads:
        def __init__(self, funcs: List[Callable], nums_of_thread: int = None):
            self.nums_of_thread = nums_of_thread
            self.funcs = funcs
            self.threads = []

        def get_nums_of_cpu(self):
            nums = multiprocessing.cpu_count()
            if self.nums_of_thread is None:
                self.nums_of_thread = nums
            return nums
```

[1]There are 31 days between 2022/10/01 to 2022/11/01

```
13          @singledispatchmethod
            def allocate_input_to_threads(self, function_input: List[Tuple]) ->
                NoReturn:
15              for i in range(self.nums_of_thread):
                    t = threading.Thread(target=self.funcs[i], args=function_input[i
                        ])
17                  self.threads.append(t)

19          @allocate_input_to_threads.register
            def _(self, function_input: tuple) -> NoReturn:
21              for i in range(self.nums_of_thread):
                    t = threading.Thread(target=self.funcs[i], args=function_input)
23                  self.threads.append(t)

25          @allocate_input_to_threads.register
            def _(self, function_input: float) -> NoReturn:
27              for i in range(self.nums_of_thread):
                    t = threading.Thread(target=self.funcs[i], args=(function_input
                        ,))
29                  self.threads.append(t)

31          def start_calc(self) -> NoReturn:
                # start sending
33              for i in self.threads:
                    i.start()
35
                # wait for all thread finish
37              for i in self.threads:
                    i.join(timeout=360)
```

Listing 1.6: MultiThread Object

In this class, "allocate_input_to_threads()" will allocate different types of inputs into different threads. For example,if the input type is "tuple", then "_(self, function_input: tuple) - NoReturn:" is called. Calculation will start by calling function "start_calc()". But there will be no return for this function. Implementations of multi-thread for different requests can be found in file "multi_thread.implement", in this SDK, only multi-thread for extraction requests is implemented. List 1.7 shows the code of this implementation. Also, the implementation of multi-thread is encapsulated in function "download_market_depth_data_file()" in file "market_data.operations". This function is a combination of "multi-thread implement" and a "date-spliter" which can separate the input period into several sub-periods automatically, and the minimum time unit for separation is daily. Users do not need to construct the "OnDemandExtractioner class" and "Multi-

ThreadsImp" class when downloading data. Next section (section 1.4) gives an example of using this function.

```
    class ExtractionImp(MultiThreadsImp):

        def __init__(self, extractioners: List[OnDemandExtractioner]):
            self.extractioners = extractioners
            self.nums_of_threads = len(extractioners)

        def get_locations(self, token=None):
            token = token if token else _get_token()

            funcs = [extractioner.get_location for extractioner in self.
                extractioners]
            threads = MultiThreads(funcs=funcs, nums_of_thread=self.
                nums_of_threads)
            threads.allocate_input_to_threads((token,))
            threads.start_calc()
            locations = [ele.location for ele in self.extractioners]

            return locations

        def get_job_ids(self, token=None):
            token = token if token else _get_token()

            funcs = [extractioner.get_job_id for extractioner in self.
                extractioners]
            threads = MultiThreads(funcs=funcs, nums_of_thread=self.
                nums_of_threads)
            threads.allocate_input_to_threads((token,))
            threads.start_calc()
            job_ids = [ele.job_id for ele in self.extractioners]
            return job_ids

        def get_bodys(self, token=None):
            token = token if token else _get_token()

            funcs = [extractioner.get_body for extractioner in self.
                extractioners]
            threads = MultiThreads(funcs=funcs, nums_of_thread=self.
                nums_of_threads)
            threads.allocate_input_to_threads((token,))
            threads.start_calc()
            bodys = [ele.body for ele in self.extractioners]

            return bodys

        def save_files(self, output_file_names: List[str], token=None):
            token = token if token else _get_token()

            funcs = [extractioner.save_output_file for extractioner in self.
                extractioners]
            threads = MultiThreads(funcs=funcs, nums_of_thread=self.
                nums_of_threads)
            funcs_inputs = [(name, token) for name in output_file_names]
            threads.allocate_input_to_threads(funcs_inputs)
            threads.start_calc()
            files_dir = [extractioner.output_file_path for extractioner in self.
                extractioners]

            return files_dir
```

Listing 1.7: Extraction Implementation

## 1.4 Examples for Extraction

In this section, an example for an extraction request is provided. If we want the market depth data including 2-level ask price and ask size for "CLZ24" (Ric code) between 7th Sep 2022 and 9th Sep 2022. The script can be written as list 1.8. In this script, by calling function "download_market_depth_data_file()", the extracted data will be automatically separated into daily data. The amount of maximum requests sent per time is set to 40, which means 40 daily data will be extracted at the same time.

```
# %% example for market depth for CLZ24
##  ids:
identifier = 'CLZ24'
identifier_type = IdentifierType.Ric

## dates (%Y, %M, %D, %h, %s, %ms):
query_start_date = datetime(2022, 9, 7, 0, 0, 0)
query_end_date = datetime(2022, 9, 9, 0, 0, 0)

## required content field names
ask_price = MarketDepthContentFieldNames.AskPrice
ask_size = MarketDepthContentFieldNames.AskSize
num_of_levels = 2

## save file:
logging.info('Created extractioner')
path = os.path.join(os.getcwd(), '..')
output_file_path = f'{path}/output_docs/market_depth_test.csv.gz'
logging.info('start downloading output file')
s = time.time()
download_market_depth_data_file(identifier=identifier,
                                required_fields=[ask_price, ask_size],
                                query_start_date=query_start_date,
                                query_end_date=query_end_date,
                                num_of_levels=num_of_levels,
                                output_file_path=output_file_path)
e = time.time()
print(f'Save success, use {e - s}s')
```

Listing 1.8: Extraction Example

After executing the script, the compressed data file will be saved in the "output_file_path". The extracted data will be saved into a folder ("CLZ24-2022-9") which includes two compressed files of the extracted daily data ("CLZ24-2022-09-07.csv.gz" and "CLZ24-2022-09-08.csv.gz"). Some values of the extraction result of the example is shown in Table 1.2

Table 1.2: Extraction Example Result

| RIC | Domain | Date-Time | L1-AskPrice | L1-AskSize | L2-AksPrice | L2-AskSize |
|---|---|---|---|---|---|---|
| CLZ24 | Market Price | 2022-09-07T00:00:02.104727306Z | 71.74 | 1 | 71.75 | 2 |
| CLZ24 | Market Price | 2022-09-07T00:00:02.240015939Z | 71.74 | 1 | 71.75 | 2 |
| ... | ... | ... | ... | ... | ... | |
| CLZ24 | Market Price | 2022-09-07T00:05:12.936494198Z | 71.56 | 5 | 71.57 | 5 |
| ... | ... | ... | ... | ... | ... | |
| CLZ24 | Market Price | 2022-09-08T00:00:15.028921134Z | 69.34 | 3 | 69.35 | 2 |
| ... | ... | ... | ... | ... | ... | |
| CLZ24 | Market Price | 2022-09-08T23:58:50.852610474Z | 69.37 | 1 | 69.38 | 1 |

[1] The table only shows selected data from the extracted results.

## 1.5 How to Add Extraction Request

Refinitiv has provided many different types of extraction templates, such as "Intraday Summary extraction", "Market Depth extraction", "End of Day extraction", "Time and Sales extraction", "Time Series extraction" and etc. However, in this SDK, only "Market Depth extraction" and "Intraday Summary extraction" have been implemented. But there is a simple way to add new extraction. In this SDK, each extraction is an object "Extractioner" inherited the supper class "OnDemandExtractioner". When adding a new extraction, only the input and the function "get_body()" need to be rewritten. List 1.9 shows the contribution of "Market Depth Extractioner".

```
    class TickHistoryMarketDepthExtractioner(OnDemandExtractioner):
2       def __init__(self, identifier_list: InstrumentIdentifierListBase,
                     market_depth_content_field_names: List[
                         MarketDepthContentFieldNames],
4                    condition: TickHistoryMarketDepthCondition,
                     extraction_type: ExtractionTypes = ExtractionTypes.
                         ExtractRaw):
6           super().__init__()
            self.condition = condition
8           self.identifier_list = identifier_list
            self.market_depth_content_field_names =
                market_depth_content_field_names
10          self.extraction_data_types = RequiredExtractionDataTypes.
                TickHistoryMarketDepthExtractionRequest
            self.extraction_type = extraction_type
12
        def get_body(self) -> dict:
14          if self.body:
                return self.body
16          body = {
                self._ExtractionRequest: {
18                  self._odata_type: self._extraction_request_header + self.
                        extraction_data_types.value,
                    self._ContentFieldNames: [field_name.value for field_name in
                         self.market_depth_content_field_names],
20                  self._IdentifierList: self.identifier_list.get_dict_form('
                        Extractions.ExtractionRequests'),
                    self._Condition: self.condition.dict_form,
22              }
            }
24          self.body = body
            return body
```

Listing 1.9: Market Depth Extraction Object

In the example 1.9, the extraction object "Extractioner" is formed by 4 main parts, "identifier list", "content field names", "condition" and "extraction type". First of all, "identifier list " includes the required instruments' identifiers or some criteria for target instruments. Hence, the "identifier list" is an object of "InstrumentIdentifierListBase", which is the parent class for identifier list or criteria list. The second input "content field names" is the required data, which is different from different extractions. For example, "bid price" is allowed for "market depth" extraction, while "close bid price" or "open bid price" is available to "intraday summaries" extraction. Also, the "content field names" is an enumerate class containing all these options. After that, "condition" is a filter for the extracted data, for example, relative time or time range can be used for selecting the extracted data period. Also, the level of view and the number of level can be chosen in the condition for market depth data. Finally, "extraction type" is the returned extracted data type, "extract raw" and "extract with notes" can be selected. "extraction raw" will only return the raw extracted data, while "extract with notes" will also return a notes which includes some statistics about the instrument and the process.

The only method implemented in the class 1.9 is "get_body()", which will create the input json in the HTTP request. The format of the json can be found in Refinitiv's API reference tree, and it is different for each request. Section 1.6 provides an example about how to build the body.

Parameters | JSON Model


```
{
    "ExtractionRequest": {
        "@odata.type": "#DataScope.Select.Api.Extractions.ExtractionRequests.TickHistoryMarketDepthExtractionRequest",
        "ContentFieldNames": [
            "Ask Price",
            "Ask Size"
        ],
        "IdentifierList": {
            "@odata.type": "#DataScope.Select.Api.Extractions.ExtractionRequests.InstrumentListIdentifierList",
            "InstrumentListId": "CLZ24"
        },
        "Condition": {
            "View": "NormalizedLL2",
            "NumberOfLevels": 2,
            "MessageTimeStampIn": "GmtUtc",
            "ReportDateRangeType": "Range",
            "TimeRangeMode": "Inclusive",
            "QueryStartDate": "2023-01-21T00:00:00.000Z",
            "QueryEndDate": "2023-02-21T00:00:00.000Z",
            "DisplaySourceRIC": true
        }
    }
}
```


Figure 1.4: Market Depth API Tree Example

## 1.6 Examples for Extractioner

In this section, an example of how to build the body of a extractioner is given. To start with, Refinitiv API tree shows the format of the HTTP input json. The aim of function "get_body()" is to create the same input json. Figure 1.4 shows the example of extracting market depth data including the ask price and the ask size for "CLZ24" during 2023/1/21 to 2023/2/21.

As we can see from figure 1.4, there are three main parts for a input json, "ContentFieldNames", "IdentifierList" and "Condition", which is the same as constructors of the extraction object in the SDK. Also, the dictionary form for these three parts can be built separately and automatically by calling ".dict_form" for each class. For example, the dictionary form of condition can be built by "self.condition.dict_form". Finally, the key for this json has already written in the parent class "OnDemandExtractioner" by putting a "_" in front of these keys.

Hence, for example, the dictionary of condition can be built by "self._Condition: self.condition.dict_form". Other keys can be created by using the similar way. List 1.10 shows the dictionary of the body part of market depth extraction request. And the corresponding output is given in List 1.11.

```
body = {
    self._ExtractionRequest: {
        self._odata_type: self._extraction_request_header + self.
            extraction_data_types.value,
        self._ContentFieldNames: [field_name.value for field_name in self.
            market_depth_content_field_names],
        self._IdentifierList: self.identifier_list.get_dict_form('
            Extractions.ExtractionRequests'),
        self._Condition: self.condition.dict_form,
    }
}
```

Listing 1.10: Body Part Example

```
body = {'ExtractionRequest':
        {'@odata.type': '#DataScope.Select.Api.Extractions.
            ExtractionRequests.TickHistoryMarketDepthExtractionRequest',
         'ContentFieldNames': ['Ask Price',
                               'Ask Size'],
         'IdentifierList':
             {'@odata.type': '#DataScope.Select.Api.Extractions.
                 ExtractionRequests.InstrumentIdentifierList',
              'InstrumentIdentifiers':
                  [{'Identifier': 'CLZ24', 'IdentifierType': 'Ric'}],
              'ValidationOptions': {'AllowHistoricalInstruments': True},
              'UseUserPreferencesForValidationOptions': False},
         'Condition': {'QueryEndDate': '2023-02-21T12:00:00.000Z',
                       'QueryStartDate': '2023-01-21T12:00:00.000Z',
                       'NumberOfLevels': 2,
                       'View': 'NormalizedLL2',
                       'DisplaySourceRIC': True,
                       'ExtractBy': 'Ric',
                       'MessageTimeStampIn': 'GmtUtc',
                       'ReportDateRangeType': 'Range',
                       'SortBy': 'SingleByRic'}
         }
    }
```

Listing 1.11: Body Part Example Output

After successfully get the body part, the new extractioner is added, this new extractioner class can implement all methods in the parent class ("OnDemandExtractioner") including "save_output_file()".Finally, the extracted data file of all child classes can be downloaded by calling "save_output_file()".

## 1.7 A GPU programming for extracting the RND

To process the large dataset efficiently, we utilize GPU (Graphics Processing Unit) programming. Unlike the CPU (Central Processing Unit), which is designed for a broad range of distinct tasks, the GPU is specifically optimized for executing a large number of simpler, parallel tasks. Figure 1.5 illustrates the architectures of both the CPU and GPU

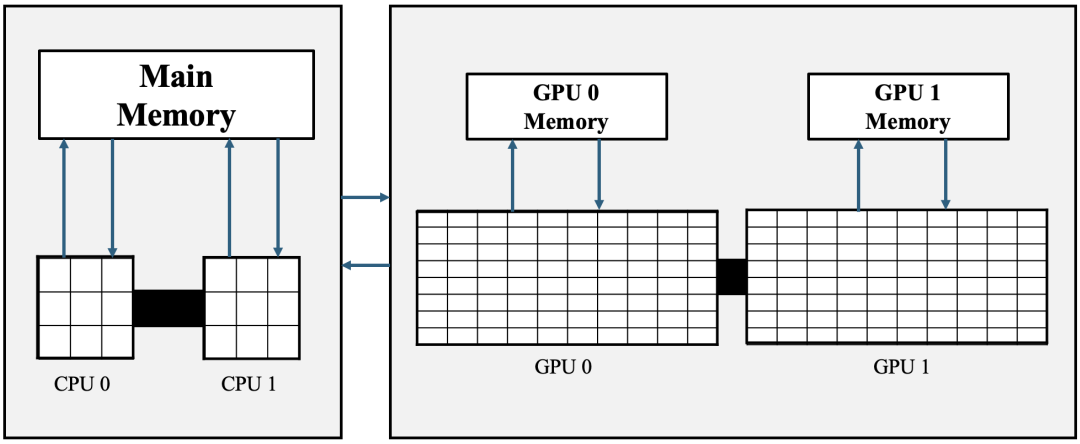


Figure 1.5: CPU and GPU Architecture

Although modern CPUs have multiple cores, the number is still relatively small compared to GPUs. For instance, the AMD EPYC 9754 CPU has 128 cores[1], whereas the Nvidia A30 GPU has 3,804 cores[2]. Consequently, GPUs can process many more tasks simultaneously than the CPU.

For our research, we aim to calculate the implied volatilites based on the market-observed prices. Therefore, we utilized the method proposed by Breeden and Litzenberger [1978] to calculate the risk-neutral density:

[1]Source: https://www.amd.com/en/products/cpu/amd-epyc-9754
[2]CUDA cores, source: https://www.pny.com/nvidia-a30

$$\frac{\partial^2 C(t,T,K)}{\partial K^2} = e^{-r(T-t)} g(S_T)$$

We use finite difference method to estimate the partial differential terms. For determining the implied volatility, we applied Newton-Raphson method (see Eq. (1.1)) to find the optimal solution to Eq. (1.2).

$$vol_{n+1} = vol_n - \frac{BS(vol_n)}{BS'(vol_n)} \tag{1.1}$$

$$\begin{aligned} \boldsymbol{vol}^* &= \underset{\boldsymbol{vol}}{arg\,min} ||\boldsymbol{P} - \boldsymbol{BS}(\boldsymbol{vol}_0)||^2 \\ s.t.\, vol &> 0 \end{aligned} \tag{1.2}$$

where $BS(\cdot)$ denotes the Black-Scholes formula (Black and Scholes [1973]). $BS'(vol_n)$ represents the Vega for the option. $P$ is the option price observed from the market. Since GPUs are designed exclusively for numerical computations, the raw data (see Tab. 1.3) must first be processed into detailed option data (see Tab. 1.4). Subsequently,this data is transferred to the GPU for the computation of the risk-neutral density. Figure 1.6 illustrates how data is processed by the GPU and CPU. Compared with CPU computing which takes days to process the data, the GPU takes 4 hours to finish calculate the data for a month.

| #RIC | Date-Time | Bid Price | Bid Size | Ask Price | Ask Size |
|---|---|---|---|---|---|
| TY1205R3 | 2013-01-28T15:01:39.356235000Z | 0.0625 | 4 | | |
| TY1205R3 | 2013-01-28T15:01:39.356235000Z | 0.0625 | | 0.09375 | 100 |
| TY1205R3 | 2013-01-28T15:05:13.016758000Z | 0.0625 | | 0.078125 | |
| TY1205R3 | 2013-01-28T15:06:51.578188000Z | 0.0625 | | 0.078125 | |
| TY125O3 | 2013-01-15T14:28:03.383112000Z | | | 0.015625 | |
| ... | ... | ... | ... | ... | ... |
| TYM3 | 2013-01-31T23:40:36.997409000Z | 130.046875 | 14 | 130.0625 | 4 |
| TYM3 | 2013-01-31T23:40:36.997409000Z | 130.046875 | 14 | 130.078125 | 23 |
| TYM3 | 2013-01-31T23:40:36.997409000Z | 130.046875 | 14 | 130.078125 | 21 |
| TYM3 | 2013-01-31T23:40:36.997409000Z | 130.046875 | 17 | 130.078125 | 21 |

Table 1.3: Raw Data

| #RIC | Date-Time | Bid Price | Bid Size | Ask Price | Ask Size | Contract Type |
|---|---|---|---|---|---|---|
| TY10025B3 | 2023-01-20T13:20:28.367299648Z | 15.0 | 1.0 | 15.109375 | 1.0 | Option |
| TY10025B3 | 2023-01-20T13:20:30.643362911Z | 14.984375 | 1.0 | 15.109375 | 1.0 | Option |
| TY10025B3 | 2023-01-20T13:20:33.058488285Z | 15.0 | 1.0 | 15.109375 | 1.0 | Option |
| TY10025B3 | 2023-01-20T13:20:33.226949783Z | 14.984375 | 1.0 | 15.109375 | 1.0 | Option |
| TY10025B3 | 2023-01-20T13:20:33.302647229Z | 15.0 | 1.0 | 15.109375 | 1.0 | Option |
| TY10025B3 | 2023-01-20T13:20:44.047374299Z | 15.0 | 1.0 | 15.125 | 1.0 | Option |
| TY10025B3 | 2023-01-20T13:20:45.423321752Z | 15.0 | 1.0 | 15.109375 | 1.0 | Option |
| TY10025B3 | 2023-01-20T13:20:55.515039558Z | 15.0 | 1.0 | 15.125 | 1.0 | Option |
| TY10025B3 | 2023-01-20T13:20:56.595216527Z | 15.015625 | 1.0 | 15.125 | 1.0 | Option |

Table 1.4: Option Data

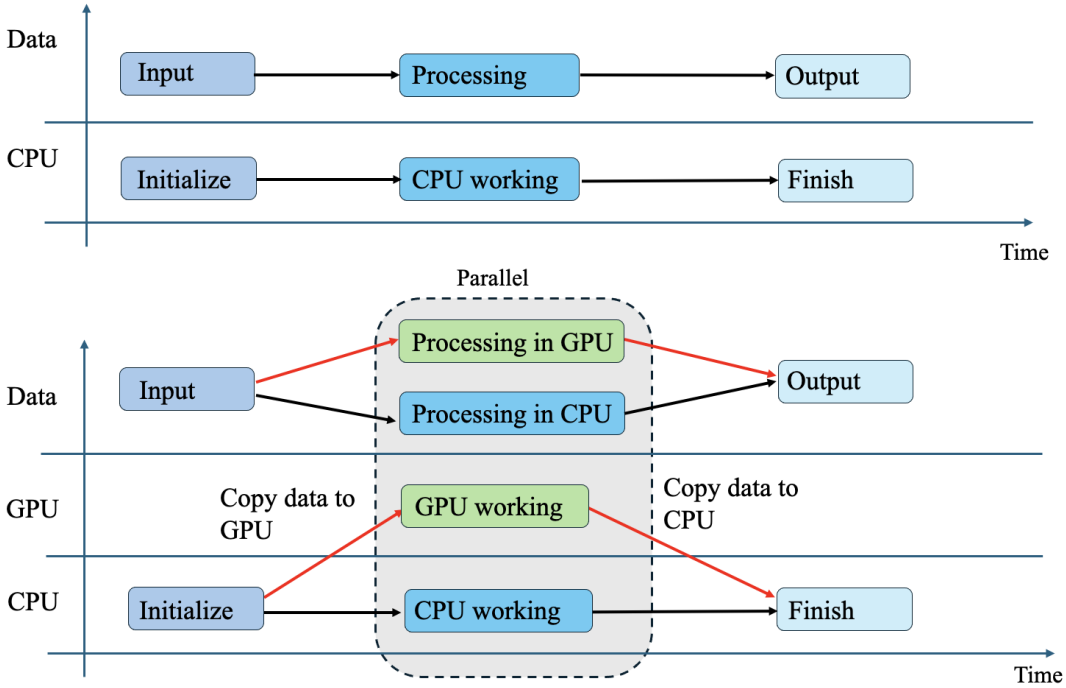


Figure 1.6: GPU Working

# References

Black, F. and M. Scholes (1973). The pricing of options and corporate liabilities. *The Journal of Political Economy*, 637–654. 22

Breeden, D. T. and R. H. Litzenberger (1978). Prices of state-contingent claims implicit in option prices. *The Joournal of Business 51*(4), 621–651. 21