Abstract
This paper presents a detailed explanation of how to access the Dell EMC™ DataIQ IndexDB using the DataIQ API to generate custom reporting of tagged data assets.

May 2021
Revisions

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>May 2021</td>
<td>Initial release</td>
</tr>
</tbody>
</table>

Acknowledgments

Author: Lynn Ragan

The information in this publication is provided “as is.” Dell Inc. makes no representations or warranties of any kind with respect to the information in this publication, and specifically disclaims implied warranties of merchantability or fitness for a particular purpose.

Use, copying, and distribution of any software described in this publication requires an applicable software license.

This document may contain certain words that are not consistent with Dell's current language guidelines. Dell plans to update the document over subsequent future releases to revise these words accordingly.

This document may contain language from third party content that is not under Dell's control and is not consistent with Dell's current guidelines for Dell's own content. When such third party content is updated by the relevant third parties, this document will be revised accordingly.

Copyright © 2021 Dell Inc. or its subsidiaries. All Rights Reserved. Dell Technologies, Dell, EMC, Dell EMC and other trademarks are trademarks of Dell Inc. or its subsidiaries. Other trademarks may be trademarks of their respective owners. [8/19/2021] [Technical White Paper] [H18782]
# Table of contents

Revisions .............................................................................................................................. 2  
Acknowledgments .............................................................................................................. 2  
Table of contents .............................................................................................................. 3  
Executive summary .......................................................................................................... 4  
1 DataIQ concepts and overview ....................................................................................... 5  
2 Getting started with simple report output ....................................................................... 6  
3 Using API report building tools ..................................................................................... 8  
4 Deeper insights with more complex report queries ....................................................... 12  
5 Using AND/OR logic capabilities to focus results ......................................................... 17  
6 Conclusion ...................................................................................................................... 20  
A Appendix ....................................................................................................................... 21  
  A.1 FastStatRequest sample Python code set ................................................................. 21  
  A.2 AND/OR sample code set ......................................................................................... 23  
B Technical support and resources ................................................................................. 24
As unstructured data amasses at an accelerated pace, it becomes more critical for the business to define the context of its data assets in terms of its own business value. The adjoining problem is how to report on that data, also within the business-defined context, and to extract that custom reporting on a scheduled basis. Dell EMC™ DataIQ™ provides toolsets that address both issues and solves the modern business need to analyze asset usage and gain relevant insights.

This paper presents a detailed explanation of how to access the DataIQ indexDB using the DataIQ API and generate custom reports featuring the API: FastStatRequest() method. Using a start-to-finish approach, this paper describes all components and steps required to construct several types of report requests. It also demonstrates the ability to incorporate the raw data output into common data science utility toolsets.

Ultimately, the business requires practical tools for extracting meaning and insights about how the organization uses data assets from the viewpoint of teams, people, projects, or processes. The ability to access these contextual categorizations using API calls opens up an entire field of custom report construction to enable data analysis and insights.
1 DataIQ concepts and overview

DataIQ dataset management elements focus on the scanning and indexing of unstructured data repositories that are accessible over standard protocols (NFS, SMB, S3), including both file and object storage. The indexing process collects roll-up statistical information about the file or object names and paths that are scanned (stored along with the name and path information). Along with the indexing function, DataIQ brings a powerful business enablement function known as Auto-tagging (see the document DataIQ Auto-tag Solutions for detailed explanations).

Auto-tagging empowers the business to categorize data assets according to arbitrary business context. In other words, the business can label unstructured datasets by organizational department or team structure, business project, grant or research ID, research group, or business process. Since business data is often spread across multiple storage platform resources, this labeling process (or tagging) allows the business to create virtual groupings of their data assets for reporting purposes.

The DataIQ API provides a secure access path to create scripted report calls which can extract both summary and detailed report output data. You can employ industry-standard toolsets (often called data science tools), to craft query-data output into customized reports which may be called on a scheduled basis.

The API is called from a Python-based wrapper library that each DataIQ instance provides.

This document details the process of building an API call to a DataIQ server with volumes that have been created, scanned, and indexed. The examples in this paper start simple and increase in complexity as the document explores the features and functionality of the API: FastStatRequest() method.
2 Getting started with simple report output

The Analyze page in the DataIQ webUI provides controls for reporting statistical information about configured volumes. It can display reports which summarize usage according to the business context tags that are applied across volumes and across storage platforms. For business stakeholders to extend and customize the report output that is generated, they can access the DataIQ indexDB and make this report content available outside of the webUI environment. Stakeholders can use the DataIQ API and the extensive report capabilities within the API library.

To demonstrate how the API might be used, the following explanation describes how to build a report query and output that is similar to what is displayed through a typical webUI Auto-tag report. These steps are intended to produce a report that is similar to this example from the DataIQ Analyze page as shown in Figure 2.

The report should have all tags for a particular category. For each tag, the webUI displays a bar graph that shows the roll-up summary usage of storage for each of those tags.

To get started, download the `claritynowapi.py` file from a DataIQ server. The DataIQ: Developer guide provides instructions for downloading the API libraries. Next, as with any IDE environment, it is necessary to import Python libraries which assist with the necessary tasks. The following block of import commands begin each of the example sections in this paper.

```python
import claritynowapi
import matplotlib.pyplot as plt  # Optional for graphing
import matplotlib as style  # Optional for graphing
import pandas as pd  # Needed for the more complex example
import numpy as np  # Needed for the more complex example
```

```python
api = claritynowapi.ClarityNowConnection('administrator', 'PWDXXXXXX', hostname='xx3.xx4.xx5.xx6', port=443, ignore_server_version=False, override_localhost=False, enable_auth=True)
```

The above two cells go through the process of importing the claritynowapi library. This library is the API access library (which is periodically updated) that you must download from the DataIQ host. An API update script downloads the most recent compiled version of the library, which is helpful if security fixes have been
Getting started with simple report output

recently incorporated into the API. However, you must download the full Python-JSON combined library for development work.

The ClarityNowConnection method creates an api object that contains useful functions (methods) to communicate directly and securely using the API. This instance initializes a secure connection to the DataIQ server in this example. The api object uses the authorization credentials and IP address of the DataIQ server to make the Get/Put calls that the FastStatRequests generates. These calls are explored later in this paper.

To test this connection, make some simple calls to DataIQ by using the basic methods of the api instance. For example, use the getTags routine to get a listing of all the tags that are present in the IndexDB for a category. In this example, the category of projects exists.

```python
myTaglist = api.getTags("projects")
```

The API call getTags takes a single argument, which is the general category of the tags to be viewed. The related call of getTag takes two arguments, which include the category and the tag which you may need to reference for other operations as shown below.

```python
myTag = api.getTag("projects", "prj1")
```

This action may seem unnecessary since you supply the tag category and the specific tag name, implying that you already know this information. Using the getTag routine enables you to query DataIQ for the critical ID value. This value is part of the tag identity that the API routines require to make report requests.

Progressing to the plural getTags call, you can iterate through the returned list to show all tag.names and tag.ids that are present in the projects category. All the labels are arbitrary, including the individual tags that depict names for various data projects, and the grouping by category. IndexDB provides these tools which highlights the powerful combination of business-context data categorization with the extended capabilities of this API library.

```python
for item in myTaglist:
    print (item.name, item.id)
```

```
prj3 241
CloudProject 242
prj3_dev 56
prj_gis1 248
prj_gis2 249
project1 250
prj2 43
prj1 44
```

Now you have validated that there is a set of tags which are applied to the scanned data in the IndexDB, and that these tags are all categorized as projects. The next step is to see what other information can be derived. One possibility is to simulate the basic Tags rollup summary in the DataIQ webUI > Analyze page. You can perform this action using the same secure, externally accessible API that you initialized previously.
3 Using API report building tools

FastStatRequest() is the basic tool for making complex tag- and volume-based queries against the IndexDB of DataIQ through the API. There are many ways to use this tool for generating different types of reports. To illustrate the basic concept, here is an example which uses the high-level SUM capability. SUM tells FastStatRequest to provide summary roll-up information rather than detailed path-level information. This example illustrates the SUM characteristic.

```python
topChartgrp = claritynowapi.FastStatRequest()
topChartgrp.resultType = claritynowapi.FastStatRequest.SUM
```

These first two lines introduce the FastStatRequest object instance and then assign a resultType of SUM. This example requests summary information as a result of the query.

```python
for item in myTaglist:
    tag = item.name
    grpSubRequest = claritynowapi.SubRequest()
    grpSubRequest.name = tag
    grpSubRequest.addTagFilter(item.id)
    topChartgrp.requests.append(grpSubRequest)
```

Using a Python for loop, this step iterates through the list of Tags that was generated into myTaglist. You must generate a separate SubRequest which contains criteria that shape the overall request. This criteria may include the unique tag ID from the myTaglist as a name for the subrequest. However, it must have another type of filter to narrow the results. By using the integrated filter addTagFilter, the tag ID can be used to limit the results to only the folders under that tag. Finally, the SubRequest is appended onto the main FastStatRequest. A more detailed explanation of the nature of `SubRequests()`, TagFilters, and other filters are provided later in this paper.

The following snippet of code is a simple byte-converter function that is added as a definition for later use and is applied to the report results. This code performs a small calculation to covert from bytes to mebibytes so that the graphs have a smaller scale.

```python
def byteconv(x):
    return x/1048576
```

Many other options exist for this type of conversion, which may be more appropriate for other reporting requirements. This optional example definition is called in a few more steps.

The api.report() call is the last step to run the report query that has been constructed. This call is also the last time that this example code interacts with the DataIQ API. All results are returned in object form, and you must parse through the results to gain the insights sought in this summary report. In the following example, topChartgrp is the specific instance of the FastStatRequest object. This instance has all the SubRequests (one for each tag) and triggers a full object response captured by ChartResults.

```python
ChartResults = api.report (topChartgrp)
```
The API has delivered results to our shell interface query, and now you must parse them out to gain a more readable report output format. The following code snippet creates a Python list object. Then, it iterates through the requests (each SubRequest is granted its own results list object), and extracts the tag name and the total byte count. These values are appended to the chartDF list, which creates a list of lists.

```python
chartDF = []
for x in range(len(ChartResults.requests)):
    a = [ChartResults.requests[x].name,
         ChartResults.requests[x].results[0].bytes,
         ChartResults.requests[x].results[0].value
    ]
    chartDF.append(a)
```

Moving into the realm of data sciences, it is helpful to use the Pandas dataframe toolset. Importing the ChartResults values (in the form of a list of lists) into a dataframe makes it possible to do the following:

- Assign column names
- Make the conversion from bytes to mebibytes (base 2)
- Indicate which column is the index
- Get a simple summary output by calling the name of the dataframe shown as follows

```python
dfout = pd.DataFrame(chartDF, columns=['Name', 'Bytes', 'MiBValue'])
dfout['MiBValue'] = dfout['Bytes'].apply(byteconv)
dfout = dfout.set_index('Name')
```

```
dfout

<table>
<thead>
<tr>
<th>Name</th>
<th>Bytes</th>
<th>MiBValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>CloudProject</td>
<td>42777724</td>
<td>40.796017</td>
</tr>
<tr>
<td>prj1</td>
<td>36619852</td>
<td>34.923412</td>
</tr>
<tr>
<td>prj2</td>
<td>416781793</td>
<td>397.474092</td>
</tr>
<tr>
<td>prj3</td>
<td>184843</td>
<td>0.176280</td>
</tr>
<tr>
<td>prj3_DEV</td>
<td>6429700</td>
<td>6.131840</td>
</tr>
<tr>
<td>prj_gis1</td>
<td>3439697</td>
<td>3.280351</td>
</tr>
<tr>
<td>prj_gis2</td>
<td>19287297</td>
<td>18.393800</td>
</tr>
<tr>
<td>project1</td>
<td>11014375</td>
<td>10.504127</td>
</tr>
</tbody>
</table>
```
The final step for a report output is to add a charted summary. Using one a common charting toolset and applying some basic setup, you can generate a bar chart from the mebibyte column. This chart shows the total on-disk usage or consumption of space for each tag in the category group of **projects**.

```python
plt.figure()
plt.style.use('seaborn-darkgrid')
plt.title('Breakout for Category: projects')
dfout['MibValue'].plot.barh(figsize = (12,8), fontsize = 12)
plt.show()
```

More information about usage of matplotlib tools is available at the [Matplotlib project site](https://matplotlib.org). The above example code snippet calls the `plot` library, assigns a chart style type, and assigns a chart title. The final line references the dataframe `dfout` and tells matplotlib to use the `MibValue` column for the bar chart. The `dfout` dataframe `plot` method calls for a bar graph that is horizontal (barh). See Figure 3 for an example.

![Breakout for Category: projects](image)
As a validation check, look at the Analyze page results (Figure 4) for the same category of projects. Comparing the two output views (API report and webUI > Analyze page) verifies that the data results are similar except for the sort order.

![Figure 4: Analyze page results](image)

In this way, a coded script could be used to generate a scheduled report. The script could be added to a cron-job and scheduled for a weekly output with the report emailed to the appropriate manager for review. In this regard, you can use DataIQ to gain data insight through scheduled reporting.
Deeper insights with more complex report queries

The DataIQ API (claritynowapi.py) contains **FastStatRequest**, a multitool class that you can use to gain insights by constructing specific logic-oriented queries. This class has many options and native methods for qualifying a query to the IndexDB. The following examples explore common ways to use this query tool.

FastStatRequest enables building a complex group of queries against the IndexDB. This ability makes it possible to more closely examine results based on in-context data categorization. For example, you may be interested in assessing disparate folder groupings which may all share the same tag. Rollup summary usage and applied ROI costs may also be of interest from a business point of view.

As in the previous example, you must create an instance of the FastStatRequest. Here, an instance is called and assigned to **myrequest**. First, you define the resultType (the result type that you require from the IndexDB) as ALL_PATHS. This type indicates to the API process that you require a PathResults type of result list. As shown in the previous example, the other most common resultType is SUM (capitalization of the resultType is deliberate and required).

```python
myrequest = claritynowapi.FastStatRequest()
myrequest.resultType = claritynowapi.FastStatRequest.ALL_PATHS
```

Creating a **myrequest** object initializes a clean instance of FastStatRequest. This instance is similar to a package template. You must customize this template with all qualifying conditions and criteria which compose the specific request.

After you create the overall FastStatRequest and define the resultType, you can add criteria for the search query to narrow down ALL_PATHS. These criteria are referred to as filters in DataIQ. Start by building a SubRequest which allows you to stack multiple request actions by appending them.

```python
mySubrequest1 = claritynowapi.SubRequest()
```

It is possible to add a name or label to the SubRequest which is useful when debugging and calling these objects manually. There are two simplified filters which are designed to do most of the hard work for you: `addVolumeFilter` and `addTagFilter`. Here is an example of how to add a simple `VolumeFilter` to a subRequest. An example volume, which exists on a test instance of Dataiq, is labeled **NFS_Boston_corp**. Use the volume name within quotes for the `addVolumeFilter`.

```python
mySubrequest1.name = 'Sub1 Volumes'
mySubrequest1.addVolumeFilter("NFS_Boston_corp")
```

The SubRequest is then appended onto the original FastStatRequest.

```python
myrequest.requests.append(mySubrequest1)
```

The last four lines of code yield a path result of the top-level volume or path for the volume that is labeled **NFS_Boston_corp** as defined in DataIQ.

---

**Note:** The previous example SubRequest only shows the existence and usage of the `addVolumeFilter` option. It does not correspond to the main task in this section which is reporting on folder-trees which have been tagged. This topic is covered in section 5.

Since the intent is to query for data paths which have been labeled with a specific tag, the DataIQ API provides a powerful combination of PathResult (ALL_PATHS) and tag filtering. This next example revisits the
earlier tag category of **projects**. For instance, you would like to know what folder-trees have been tagged **prj2** across any volumes (storage platform shares). The query might use a sequence that is similar to the following and builds from the `FastStatRequest()` that was constructed at the beginning of this section.

```python
mySubrequest2 = claritynowapi.SubRequest()
mySubrequest2.name = 'project tags'
tag1 = api.getTag("projects", "prj2")
mySubrequest2.addTagFilter (tag1.id)
myrequest.requests.append(mySubrequest2)
```

The SubRequest instance is created and given a name. Then, the `getTag` method is called to obtain the tag information for the specific tag that requires analysis. Finally, the tag ID is sent to the `addTagFilter` as part of the SubRequest.

```python
for r in myreport:
    for x in r.paths:
        print(x.path, x.size, x.cost)
cost_series.append(x.cost)
```

The result list of this call for any paths labeled with tag **prj2** yields three paths. These paths are spread across two different volumes, which represent two different PowerScale or Isilon clusters. These volumes are labeled `NFS_Boston_corp` and `NFS_Madrid`. Also, the query demonstrates the ability to pull the roll-up summary size of the tagged folder tree and the TCO cost distribution as applied to each of these folder structures.
Deeper insights with more complex report queries

**Note:** This action can only be performed if a TCO is assigned to the volume by the business.

However, more detail about these results is available within this same result object. Details about the output rows shown above are contained as something similar to data fields for each result row. You can extract data from these fields by referencing the `info` data container within the result object. The following example is a subset of this extra detail. See the appendix A for full listing of data detail options.

```python
subTTLcosts = np.array(cost_series)  # costs per TiB per month
```

```python
for r in myreport:
    print([(path.info.dirCount,  
             path.info.fileCount,  
             path.info.fileType,  
             path.info.name,  
             path.info.path,  
             path.info.size)  
           for path in r.paths])

[(0, 574, 'FOLDER', 'prj2', '/NFS_Boston_corp/Project/prj2', 11628963),  
 (0, 163, 'FOLDER', 'repo2', '/NFS_Boston_corp/repo2', 388619640),  
 (0, 600, 'FOLDER', 'prj2_pln', '/NFS_Madrid/data/Project/prj2_pln', 16533190)]
```

You may apply a tag to multiple folders or paths across several volumes as a way of aggregating related worksets that are spread across different storage file structures. As noted earlier, there may be several rows of return results to parse. Printing these data-fields demonstrates that you can gather these items in a list format which can prove useful for data analysis.

In the following example, an empty Python list is initialized to contain and organize the result information instead of printing to screen. This action contains the list of fields for each path result returned, creating a list of lists for further analysis.

```python
query_d = []
for r in myreport:
    for path in r.paths:
        a = [path.info.dirCount,  
             path.info.fileCount,  
             path.info.fileType,  
             path.info.name,  
             path.info.path,  
             path.info.size]
        query_d.append(a)
```

As noted previously, you can call on the Python Pandas dataframe toolset to further organize the result data for more specific data analysis.
Below, a dataframe is created and column labels are assigned. Again, a small function is created for roughly converting bytes to mebibytes so the output is more easily interpreted.

```python
df = pd.DataFrame(query_d)
df.columns = ["dirCount", "FileCount", "FileType", "FolderName", "TagPath", "SUMdataSize"]

def byteconv(x):
    return x/1048576

#Converting Bytes values to MiB for the whole column
df['SUMdataSize'] = df['SUMdataSize'].apply(byteconv)

#add column of overall path subtotals for this category and tag
df['AggCost'] = subTTLcosts
df['Pct_of_TTLcost'] = 100 * (df['AggCost'] / subTTLcosts.sum())
df['Pct_of_TTLdata'] = 100 * (df['SUMdataSize'] / df.SUMdataSize.sum())
```

Printing `df` (the name of the data frame) generates the following table. The added calculated columns of `Pct_of_TTLcost` and `Pct_of_TTLdata` respectively show the derived values per row of percent-of-total cost of the data which shares the same tag. The columns also show the percent-of-total data size for all data that shares the same tag.

```
<table>
<thead>
<tr>
<th>dirCount</th>
<th>FileCount</th>
<th>FileType</th>
<th>FolderName</th>
<th>TagPath</th>
<th>SUMdataSize</th>
<th>AggCost</th>
<th>Pct_of_TTLcost</th>
<th>Pct_of_TTLdata</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>FOLDER</td>
<td>prj2</td>
<td>/NFS_Boston_corp/Project/prj2</td>
<td>11.090243</td>
<td>0.034867</td>
<td>2.700896</td>
<td>2.79018</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>FOLDER</td>
<td>repo2</td>
<td>/NFS_Boston_corp/repo2</td>
<td>370.616570</td>
<td>1.166859</td>
<td>92.259229</td>
<td>93.24295</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>FOLDER</td>
<td>prj2_pln</td>
<td>/NFS_Madrid/data/Project/prj2_pln</td>
<td>15.767279</td>
<td>0.090933</td>
<td>7.039875</td>
<td>3.96607</td>
</tr>
</tbody>
</table>
```

```python
plt.figure();
plt.style.use('seaborn');
df.plot.bar(y=['FileCount', 'SUMdataSize'], figsize = (14,6), fontsize=12, rot=15, title="Tag 'prj2' Detail");
plt.show(block=True);
```
Deeper insights with more complex report queries

Like the previous example, matplotlib plots a graph of columns from the dataframe. In this case, you can render a graph which compares two columns: file count for each tagged path, and total file size for each tagged path.

![Graph comparing file count and total file size](image)

Also like the previous example, by accessing specific columns from the original dataframe, you can use other sum utilities that are integrated in the dataframe.

```python
defSummary = pd.DataFrame(columns = ["CategoryFileCountTTL", "CategoryDataSizeTTL", "CategoryTTLcost"])
dfSummary['CategoryFileCountTTL'] = pd.Series(df.FileCount.sum())
dfSummary['CategoryDataSizeTTL'] = pd.Series(df.SUMdataSize.sum())
dfSummary['CategoryTTLcost'] = pd.Series(subTTLcosts.sum())
dfSummary
```

Printing the name `dfSummary` for this new dataframe yields the final summary output.

```
<table>
<thead>
<tr>
<th>CategoryFileCountTTL</th>
<th>CategoryDataSizeTTL</th>
<th>CategoryTTLcost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1337</td>
<td>397.474092</td>
</tr>
</tbody>
</table>
```
5 Using AND/OR logic capabilities to focus results

The DataIQ API enables complex filtering requests that are based on AND/OR logic options. AND/OR logic applies to conditional queries in which a condition must be true to return a result from the query. This practice can be confusing when you try to apply AND/OR conditions in terms of the output results that you seek.

To further describe the AND condition, to obtain a good result for a question, this must be true AND that must be true. For example, you would like to find a folder or path that is tagged in a category AND only exists on a specific volume. In this way, the result is narrowed significantly.

An OR condition allows more of an aggregation approach where multiple conditions could be true. With this condition, this can be true OR that can be true, and either case returns a good result. For example, you would like to find a folder or path that has this tag OR that tag. In either case, a tagged path is returned. This condition is useful for getting a summary of all tags within a category, for instance.

Exploring the AND capability a bit further, consider this sequence of TagFilter appends.

```python
tag1 = api.getTag("cleanup", "Temp")
mySubrequest1.addTagFilter (tag1.id)
```

The first addTagFilter looks for data that is tagged by the label Temp from the cleanup category.

```python
tag2 = api.getTag("projects", "CloudProject")
mySubrequest1.addTagFilter (tag2.id)
```

A second addTagFilter is added which looks for data tagged by the label CloudProject from the projects category. Then, the SubRequest object mySubrequest1, which contains both filter requests, is appended to the primary FastStatRequest myrequest.

```python
myrequest.requests.append(mySubrequest1)
myresults = api.report (myrequest)
print (myresults.requests[0].results[0].paths)
```

/PublicS3Cloud/Temp
Using AND/OR logic capabilities to focus results

The output yields a single path result of /PublicS3Cloud/Temp which matches both criteria and excludes all other paths. By validating this output in the webUI, you can see the same path from the S3 volume, and in the lower-right pane, both tags are visible. The AND condition found exactly one match for this criteria.

A single row is returned in this case, which you can verify in the Analyze page of DataIQ on the same folder. This result shows that both criteria are met and all other rows are excluded. Therefore, it is an AND condition.

To summarize, the integrated SubRequest functions of addTagFilter are considered to be an AND condition where both cases must be true to yield a result. Similarly, when the SubRequest functions of addTagFilter and addVolumeFilter are both used for the same SubRequest object instance, that also creates an AND condition. In this condition, the tagged folder must exist on the specified volume for the request to be true and to yield a result list.

However, consider a scenario when two separate SubRequest objects are initialized that have selection criteria (filters) added to them. Also, both SubRequests are stacked onto the FastStatRequest object using the requests.append function. In this case, these actions operate as an OR condition, meaning that the condition is true if SubRequest1 is true OR SubRequest2 is true.
For example, SubRequest1 might filter for all paths that have been tagged as \texttt{prj2}. SubRequest2 might filter all rows that exist on a specific volume. These criteria are treated as an OR condition where results may be yielded for either condition. This result is shown in the example below. To get the full set of results, a similar nested loop is created which has an extra loop level to get the results from all SubRequest objects. This practice also shows that each SubRequest object is paired with a result set.

```python
myrequest = claritynowapi.FastStatRequest()
myrequest.resultType = claritynowapi.FastStatRequest.ALL_PATHS

mySubrequest1 = claritynowapi.SubRequest()
mySubrequest1.name = 'Sub1 Volumes'
mySubrequest1.addVolumeFilter("NFS_Boston_corp")

mySubrequest2 = claritynowapi.SubRequest()
mySubrequest2.name = 'Sub2 project tags'
tag1 = api.getTag("projects", "prj2")
mySubrequest2.addTagFilter (tag1.id)

myrequest.requests.append(mySubrequest1)
myrequest.requests.append(mySubrequest2)

myreport = myresults.requests[0].results

for p in myreport.requests:
    for r in p.results:
        print (p.name)
        for x in r.paths:
            print(x.path, x.size, x.cost)
```

Sub2 project tags
/NFS_Madrid 4110942821675 22610.1851212497
/NFS_Montreal_DevTeams 814668711114 4236.2772977928
Sub1 Volumes
/NFS_Boston_corp 460610636 1.381831907999999

The first output line is from the **SubRequest1** output which requested a specific volume \texttt{NFS_Boston_corp} out of the collection of All Volumes in the IndexDB. This result has nothing to do with the Tag query which is contained in the SubRequest2 because of the OR condition.

The second and third output lines above are from the TagFilter query that is contained in \texttt{mySubrequest2}. If the tag criteria and the volume criteria had both been arranged as an AND condition, with both being in the same SubRequest, the filtering would have found only tagged paths that exist on the volume \texttt{NFS_Boston_corp}.

As demonstrated previously in this paper, it is a best practice to verify results from complex queries to ensure that the requested output is obtained. The loop above includes a call to print the name of each SubRequest before printing its output set which verifies the results.
Conclusion

The DataIQ API provides a powerful toolset to maximize the strengths of the DataIQ indexing engine. These strengths include the ability to access data classification according to business context using the DataIQ tagging features. The API delivers the ability to remotely and securely query the DataIQ IndexDB to create customizable and schedulable reports which may be scripted and emailed. Also, the API result outputs are easily ingestible to marketplace-standard data-analysis toolsets. This ability provides a rich opportunity for gaining deeper insights into a dataset view of data assets within the business organization.
A.1 FastStatRequest sample Python code set

```python
import claritynowapi;
import numpy as np;
import pandas as pd;
import matplotlib.pyplot as plt;
from matplotlib import style;

api = claritynowapi.ClarityNowConnection('administrator', 'PWDxxxxxx',
    hostname='xx2.xx3.xx4.xx5', port=443,
    ignore_server_version=False, override_localhost=False, enable_auth=True)

myTaglist = api.getTags("projects")
for item in myTaglist:
    print (item.name, item.id)

myrequest = claritynowapi.FastStatRequest()
myrequest.resultType = claritynowapi.FastStatRequest.ALL_PATHS

mySubrequest2 = claritynowapi.SubRequest()
mySubrequest2.name = 'project tags'
tag1 = api.getTag('projects', 'prj2')
print (tag1.name)
mySubrequest2.addTagFilter(tag1.id)  # Very important step

myrequest.requests.append(mySubrequest2)
myresults = api.report (myrequest)

myreport = myresults.requests[0].results
(cost_series = []
for r in myreport:
    for x in r.paths:
        print(x.path, x.cost, x.bytes)
        cost_series.append(x.cost)

subTTLcosts = np.array(cost_series)

subTTLcosts  # costs per TiB per month

for r in myreport:
    print (((path.info.dirCount,
```
Appendix

path.info.fileCount,
path.info.fileType,
path.info.mtime,
path.info.lastScanned,
path.info.lastModified,
path.info.name,
path.info.path,
path.info.size,
path.info.uid)
for path in r.paths])

query_d = []
for r in myreport:
    for path in r.paths:
        a = [path.info.dirCount,
             path.info.fileCount,
             path.info.fileType,
             path.info.name,
             path.info.path,
             path.info.size]
        query_d.append(a)

df = pd.DataFrame(query_d)
df.columns = ["dirCount", "FileCount", "FileType", "FolderName", "TagPath", "SUMdataSize"]
df = df.set_index(["TagPath"])  #Setting the index to the Tag Path column

dfSummary = pd.DataFrame(columns = ["CategoryFileCountTTL", "CategoryDataSizeTTL", "CategoryTTLcost"])

def byteconv(x):
    return x/1048576

df['SUMdataSize'] = df['SUMdataSize'].apply(byteconv)  #Converting my Bytes to MiB

df['AggCost'] = subTTLcosts #add column of Overall path subtotals for this
category and Tag
df['Pct_of_TTLcost'] = 100 * (df['AggCost'] / subTTLcosts.sum())

df['Pct_of_TTLdata'] = 100 * (df['SUMdataSize'] / df.SUMdataSize.sum())  # Row
total data / report total data

df
plt.figure();
plt.style.use('seaborn');
df.plot.bar(y=['FileCount', 'SUMdataSize'], figsize = (14,6), fontsize=12, rot=15, title="Tag 'prj2' Detail");
plt.show(block=True);

plt.close()

dfSummary['CategoryFileCountTTL'] = pd.Series(df.FileCount.sum())
dfSummary['CategoryDataSizeTTL'] = pd.Series(df.SUMdataSize.sum())
dfSummary['CategoryTTLcost'] = pd.Series(subTTLcosts.sum())
dfSummary.CategoryFileCountTTL[0]

dfSummary

A.2 AND/OR sample code set

import claritynowapi

api = claritynowapi.ClarityNowConnection('administrator', 'PWDxxxxxx', hostname='1xx.2xx.2xx.2xx', port=443, ignore_server_version=False, override_localhost=False, enable_auth=True)

myrequest = claritynowapi.FastStatRequest()
myrequest.resultType = claritynowapi.FastStatRequest.ALL_PATHS

mySubrequest1 = claritynowapi.SubRequest()
mySubrequest1.name = 'Sub1 Volumes'
mySubrequest1.addVolumeFilter("NFS_Boston Corp")

mySubrequest2 = claritynowapi.SubRequest()
mySubrequest2.name = 'project tags'
tag1 = api.getTag("projects", "prj2")
mySubrequest2.addTagFilter (tag1.id)

myrequest.requests.append(mySubrequest1)
myrequest.requests.append(mySubrequest2)

myresults = api.report(myrequest)

for p in myresults.requests:
    for r in p.results:
        print (p.name)
        for x in r.paths:
            print(x.path, x.size, x.cost)
B Technical support and resources

Dell.com/support is focused on meeting customer needs with proven services and support.

Storage and data protection technical documents and videos provide expertise to ensure customer success with Dell EMC storage and data protection products.