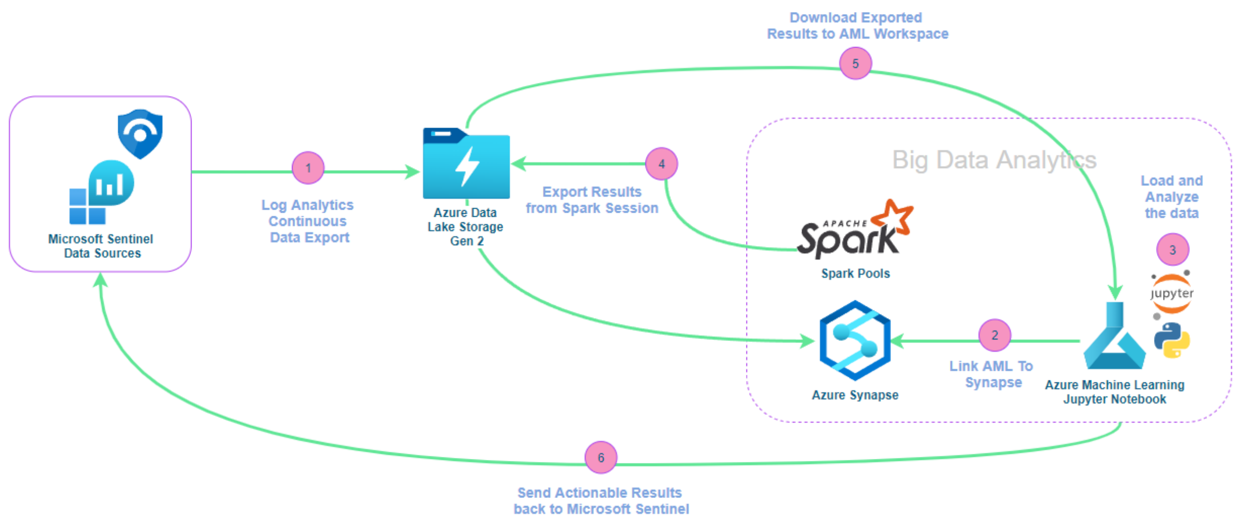
Hunting for potential network beaconing patterns using Apache Spark via Azure Synapse – Part 1

techcommunity.microsoft.com/t5/microsoft-sentinel-blog/hunting-for-potential-network-beaconing-patterns-using-apache/ba-p/2916179

November 2, 2021

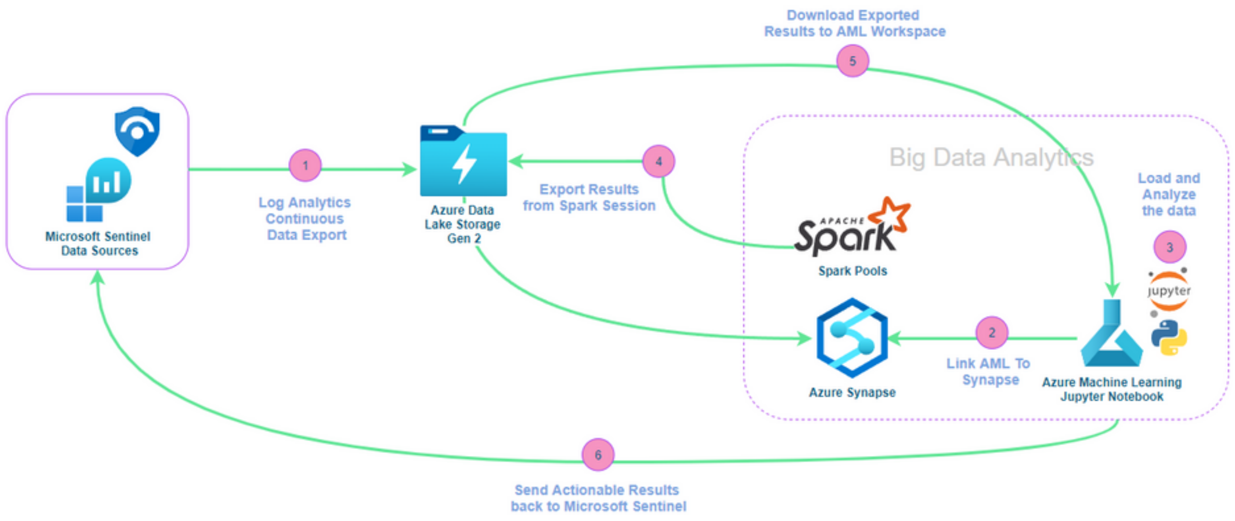

Download Exported Results to AML Workspace
5
Microsoft Sentinel Data Sources
1
Log Analytics Continuous Data Export
Azure Data Lake Storage Gen 2
4
Export Results from Spark Session
Big Data Analytics
Load and Analyze the data
3
Spark Pools
2
Link AML To Synapse
Azure Synapse
Azure Machine Learning Jupyter Notebook
6
Send Actionable Results back to Microsoft Sentinel

Introduction

We previously blogged about Detect Network beaconing via Intra-Request time delta patterns in Microsoft Sentinel using native KQL from Microsoft Sentinel. This KQL query is complex in nature and often needs to operate on very large datasets such as network firewall logs in CommonSecurityLogs table. Even after applying optimization and baselining to reduce the original datasets, sometimes such complex operations do not scale well from Log Analytics especially when you want to analyze and compare large historical datasets with the current days data. In such cases you can offload expensive data manipulation and processing operations to services outside Microsoft Sentinel and bring actionable results back to continue hunting in Microsoft Sentinel. Please review our Ignite announcement about Large Scale analytics with Azure Synapse and Microsoft Sentinel Notebooks

, In this first of 2-part blog, we will do notebook code and section walkthrough to show how you can leverage power of Apache Spark via Azure Synapse in Azure ML Notebooks to perform scalable hunting on large historical datasets to find potential network beaconing patterns. In second part, we will take a sample dataset and run the notebook on it to find potential beaconing traffic resulting from simulation of PowerShell Empire C2.

In general, the data pipeline and analysis architecture look like below.



Download Exported Results to AML Workspace
5
Microsoft Sentinel Data Sources
1
Log Analytics Continuous Data Export
Azure Data Lake Storage Gen 2
4
Export Results from Spark Session
Big Data Analytics
Load and Analyze the data
3
Spark Pools
Azure Synapse
2
Link AML To Synapse
Azure Machine Learning Jupyter Notebook
6
Send Actionable Results back to Microsoft Sentinel

Also special thanks to @Chi_Nguyen and Zhipeng Zhao for review , feedback on the blog and testing the notebook.

Apache Spark via Azure Synapse in Azure ML Notebooks

Pre-requisites

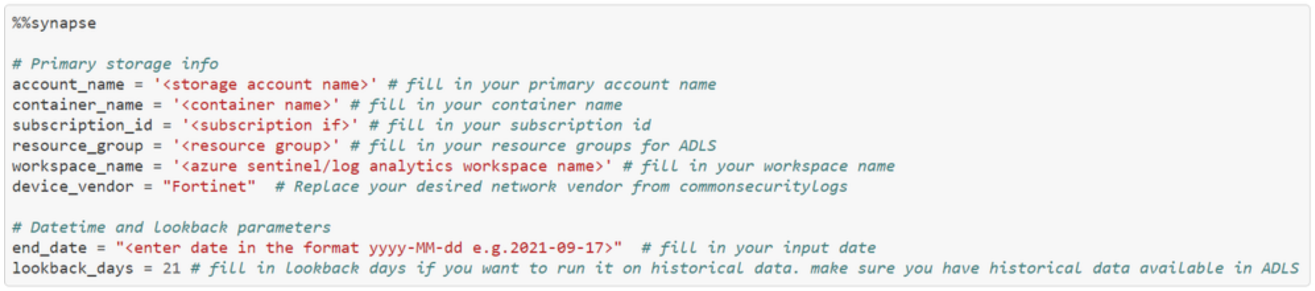
To use Apache Spark with Azure ML notebooks, you must first configure your data pipeline and connect Azure Synapse via linked service. You can check existing docs - Large-scale security analytics using Azure Sentinel notebooks and Azure Synapse integration for a step by step process and also use notebook Configurate Azure ML and Azure Synapse Analytics to set up Azure ML and Azure Synapse Analytics environment. Additionally, the notebook provides the steps to export your data from Log Analytics workspace to an Azure Data Lake Storage (ADLS) Gen2.

Loading the data

Once you have exported your logs to ADLS storage, you can connect to it and load the data required for analysis. In this case, we are going to load the current day’s dataset for analysis to find recent beaconing attempts. When you configure your data export to ADLS, it creates folder architecture per day, hour or granular 5-min buckets. Since the data is distributed across various folders, we need to programmatically generate the source path and load all the files underneath it.

Let’s start with specifying all the required parameters to generate ADLS paths. The ADLS storage file is combination of various attributes such as storage account name, container name, subscription id , resource group etc. We are also specifying additional input parameters such as device vendor, end date and lookback_days.

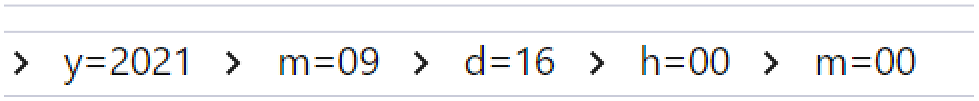
The below example cell shows where you can provide these details one time. **Note:** All the subsequent synapse cells start with ***%%synapse*** magic command. If you get an error on magic not found, run setup notebook and install required packages first.



```
%%synapse

# Primary storage info
account_name = '<storage account name>' # fill in your primary account name
container_name = '<container name>' # fill in your container name
subscription_id = '<subscription if>' # fill in your subscription id
resource_group = '<resource group>' # fill in your resource groups for ADLS
workspace_name = '<azure sentinel/log analytics workspace name>' # fill in your workspace name
device_vendor = "Fortinet"  # Replace your desired network vendor from commonsecuritylogs

# Datetime and lookback parameters
end_date = "<enter date in the format yyyy-MM-dd e.g.2021-09-17>"  # fill in your input date
lookback_days = 21 # fill in lookback days if you want to run it on historical data. make sure you have historical data available in ADLS
```

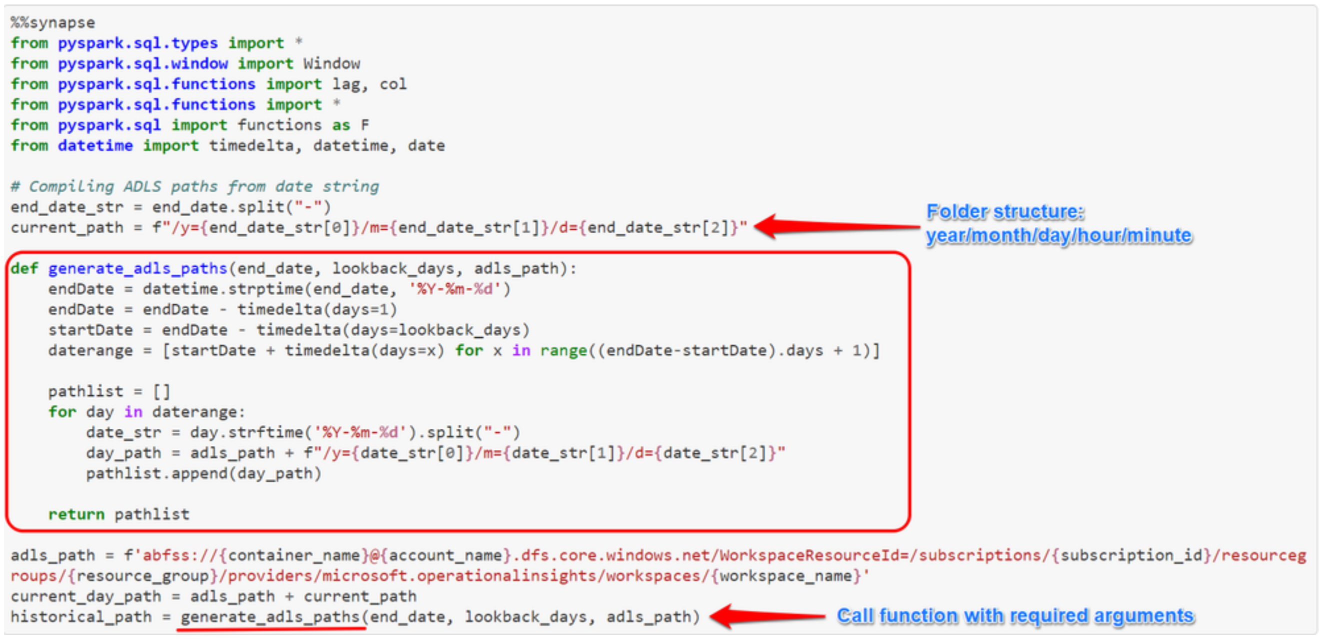
Now once we have base path, data is stored across various directories.

Below example shows sample folder structure in the format **year/month/day/hour/minute**.



```
> y=2021 > m=09 > d=16 > h=00 > m=00
```

To generate such paths programmatically, we have created a function which accepts date and lookback period as input.



```
%%synapse
from pyspark.sql.types import *
from pyspark.sql.window import Window
from pyspark.sql.functions import lag, col
from pyspark.sql.functions import *
from pyspark.sql import functions as F
from datetime import timedelta, datetime, date

# Compiling ADLS paths from date string
end_date_str = end_date.split("-")
current_path = f"/y={end_date_str[0]}/m={end_date_str[1]}/d={end_date_str[2]}"

def generate_adls_paths(end_date, lookback_days, adls_path):
    endDate = datetime.strptime(end_date, '%Y-%m-%d')
    endDate = endDate - timedelta(days=1)
    startDate = endDate - timedelta(days=lookback_days)
    daterange = [startDate + timedelta(days=x) for x in range((endDate-startDate).days + 1)]

    pathlist = []
    for day in daterange:
        date_str = day.strftime('%Y-%m-%d').split("-")
        day_path = adls_path + f"/y={date_str[0]}/m={date_str[1]}/d={date_str[2]}"
        pathlist.append(day_path)

    return pathlist

adls_path = f'abfss://{container_name}@{account_name}.dfs.core.windows.net/WorkspaceResourceId=/subscriptions/{subscription_id}/resourceg
roups/{resource_group}/providers/microsoft.operationalinsights/workspaces/{workspace_name}'
current_day_path = adls_path + current_path
historical_path = generate_adls_paths(end_date, lookback_days, adls_path)
```

Folder structure:
year/month/day/hour/minute
Call function with required arguments

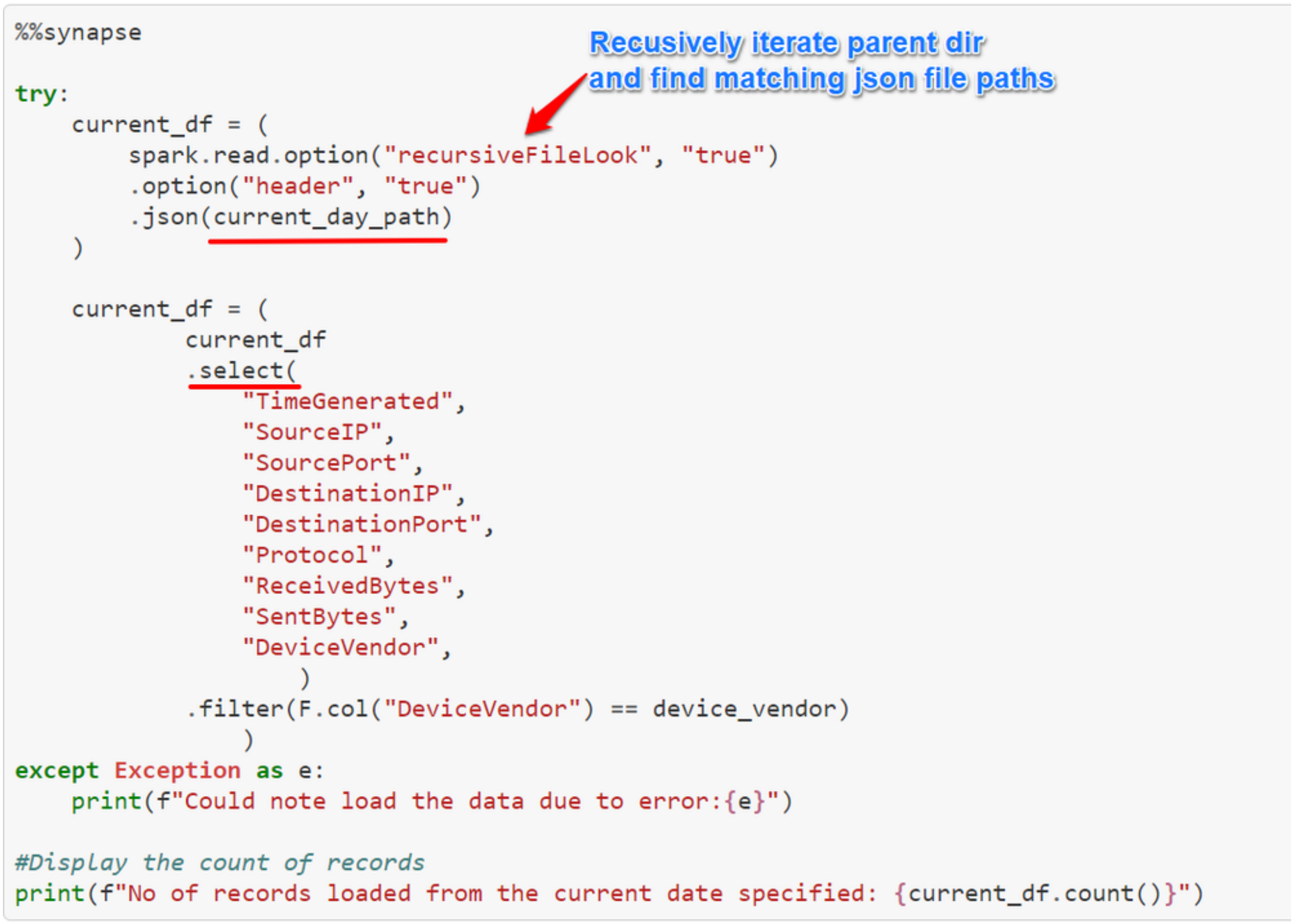
Finally, once we have generated current_day and historical_day paths, you can load the datasets as shown in below example cell. Here we have used below with additional options.

read : To read json file with additional options to keep the header and recursiveFileLook as true to recursively scan directory for matching files.

Note: recursiveFileLook is only available from Spark 3.1 so make sure to create Spark pool with 3.1 or above.

select: To select specified columns required for analysis.

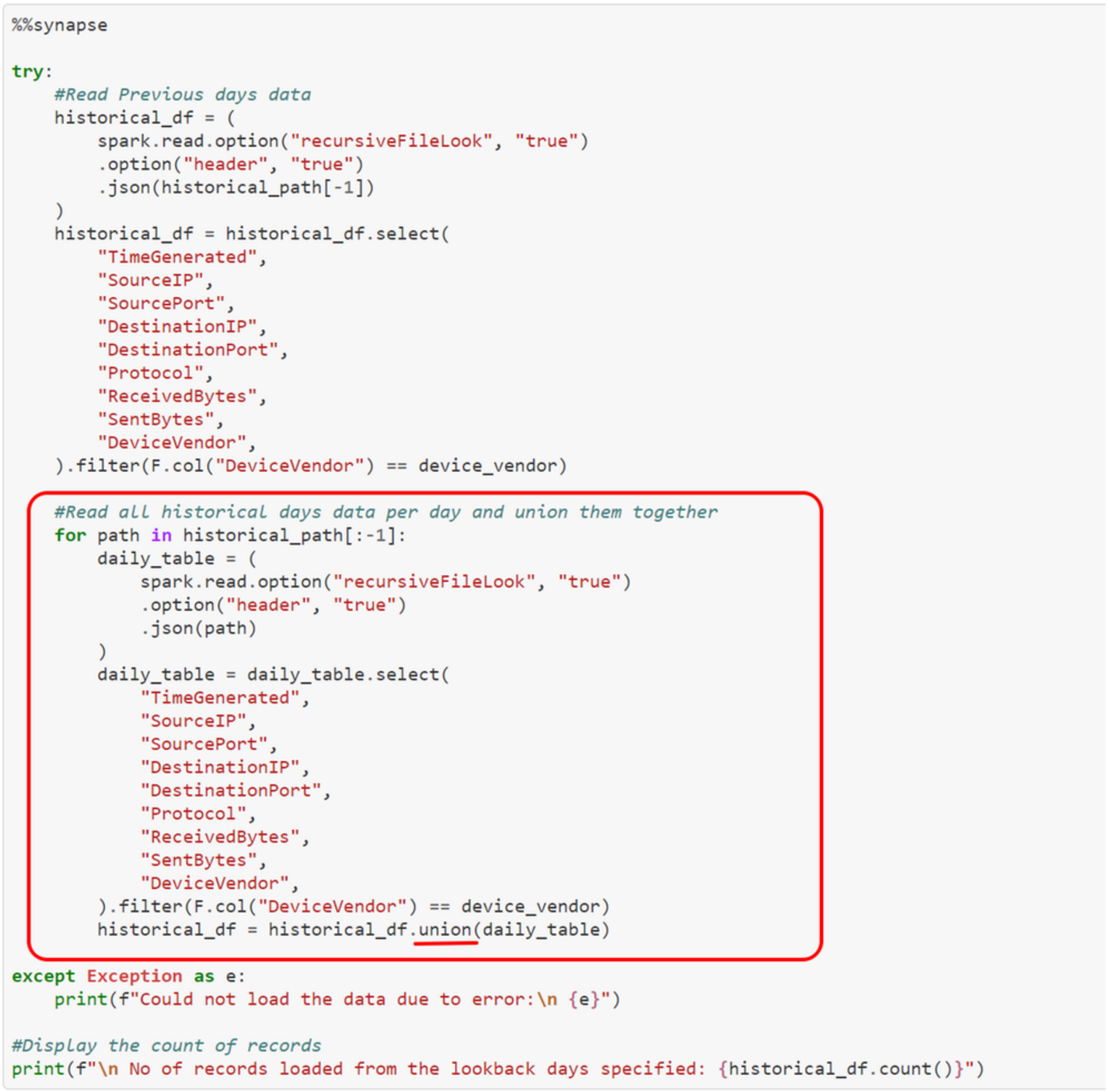
filter: To filter by column value.


Recusively iterate parent dir
and find matching json file paths

```
%%synapse

try:
    current_df = (
        spark.read.option("recursiveFileLook", "true")
        .option("header", "true")
        .json(current_day_path)
    )

    current_df = (
            current_df
            .select(
                "TimeGenerated",
                "SourceIP",
                "SourcePort",
                "DestinationIP",
                "DestinationPort",
                "Protocol",
                "ReceivedBytes",
                "SentBytes",
                "DeviceVendor",
                    )
            .filter(F.col("DeviceVendor") == device_vendor)
                )
except Exception as e:
    print(f"Could note load the data due to error:{e}")

#Display the count of records
print(f"No of records loaded from the current date specified: {current_df.count()}")
```

You can use a similar cell to load the historical dataset, where you can load individual files and union them together to load all historical data. Historical data will help in baselining the environment effectively, if you do not have historical data exported already, you can just run this on current data.

union: Union of individual days from historical days dataset.



```python
%%synapse

try:
    #Read Previous days data
    historical_df = (
        spark.read.option("recursiveFileLook", "true")
        .option("header", "true")
        .json(historical_path[-1])
    )
    historical_df = historical_df.select(
        "TimeGenerated",
        "SourceIP",
        "SourcePort",
        "DestinationIP",
        "DestinationPort",
        "Protocol",
        "ReceivedBytes",
        "SentBytes",
        "DeviceVendor",
    ).filter(F.col("DeviceVendor") == device_vendor)

    #Read all historical days data per day and union them together
    for path in historical_path[:-1]:
        daily_table = (
            spark.read.option("recursiveFileLook", "true")
            .option("header", "true")
            .json(path)
        )
        daily_table = daily_table.select(
            "TimeGenerated",
            "SourceIP",
            "SourcePort",
            "DestinationIP",
            "DestinationPort",
            "Protocol",
            "ReceivedBytes",
            "SentBytes",
            "DeviceVendor",
        ).filter(F.col("DeviceVendor") == device_vendor)
        historical_df = historical_df.union(daily_table)
except Exception as e:
    print(f"Could not load the data due to error:\n {e}")

#Display the count of records
print(f"\n No of records loaded from the lookback days specified: {historical_df.count()}")
```

```
%%synapse

try:
    #Read Previous days data
    historical_df = (
        spark.read.option("recursiveFileLook", "true")
        .option("header", "true")
        .json(historical_path[-1])
    )
    historical_df = historical_df.select(
        "TimeGenerated",
        "SourceIP",
        "SourcePort",
        "DestinationIP",
        "DestinationPort",
        "Protocol",
        "ReceivedBytes",
        "SentBytes",
        "DeviceVendor",
    ).filter(F.col("DeviceVendor") == device_vendor)

    #Read all historical days data per day and union them together
    for path in historical_path[:-1]:
        daily_table = (
            spark.read.option("recursiveFileLook", "true")
            .option("header", "true")
            .json(path)
        )
        daily_table = daily_table.select(
            "TimeGenerated",
            "SourceIP",
            "SourcePort",
            "DestinationIP",
            "DestinationPort",
            "Protocol",
            "ReceivedBytes",
            "SentBytes",
            "DeviceVendor",
        ).filter(F.col("DeviceVendor") == device_vendor)
        historical_df = historical_df.union(daily_table)

except Exception as e:
    print(f"Could not load the data due to error:\n {e}")

#Display the count of records
print(f"\n No of records loaded from the lookback days specified: {historical_df.count()}")
```

Data Preparation

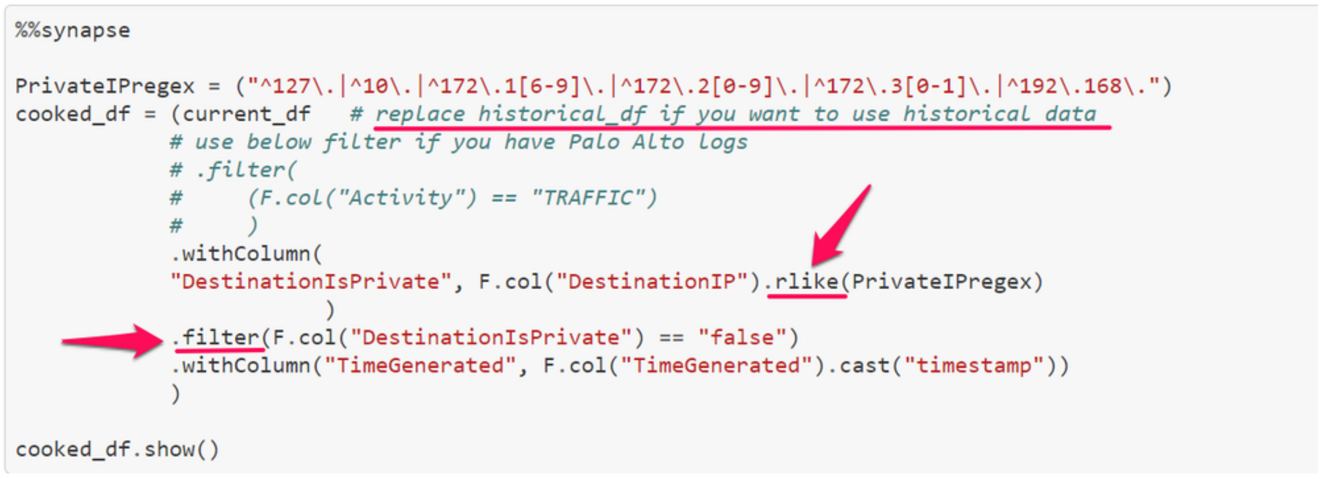
As we are looking for external network beaconing before we do the baselining, you can filter the input dataset to only IP address pair from Private to Public IP addresses. To do this, you can use regular expressions and the PySpark operator rlike. Once we have populated additional columns stating the type of destination IP, you can then filter only for public destination IP addresses.

In this cell, we have used below PySpark APIs

withcolumn : To create new column based on the condition.

rlike: To match with regular expression.

show: To show results after regex matching and filtering.



```
%%synapse

PrivateIPregex = ("^127\.|^10\.|^172\.1[6-9]\.|^172\.2[0-9]\.|^172\.3[0-1]\.|^192\.168\.")
cooked_df = (current_df   # replace historical_df if you want to use historical data
            # use below filter if you have Palo Alto logs
            # .filter(
            #     (F.col("Activity") == "TRAFFIC")
            #     )
            .withColumn(
            "DestinationIsPrivate", F.col("DestinationIP").rlike(PrivateIPregex)
                        )
            .filter(F.col("DestinationIsPrivate") == "false")
            .withColumn("TimeGenerated", F.col("TimeGenerated").cast("timestamp"))
            )

cooked_df.show()
```

Scalable Baselining across Historical Data

If you have historical dataset (lookback days from 7 to 30 depending upon the availability) loaded , you can perform baselining on it. Baselining can help reduce false positives by removing frequently occurring public destinations which are likely benign. You can take various baselining approaches to reduce input dataset.

In this notebook, we are going to use few static thresholds. You can also use dynamic logic in terms of percentages across total hosts in your environments if the thresholds are too low or high.

- **daily_event_count_threshold :** minimum number of events in a day. Default value set to 100. Any source IP addresses below threshold will be ignored.
- **degree_of_srcip_threshold:** max number of sources IP addresses per destination IP. Default value set to 25. Any destination IP addresses above this threshold will be ignored.

Once we identified both source IP and destination IPs from the baselining, you can further join this with the original current dataset to find beaconing patterns.

Below cell shows such baselining operations along with spark APIs.

groupBy : Groups dataframe using specified columns so we can run aggregations on them.

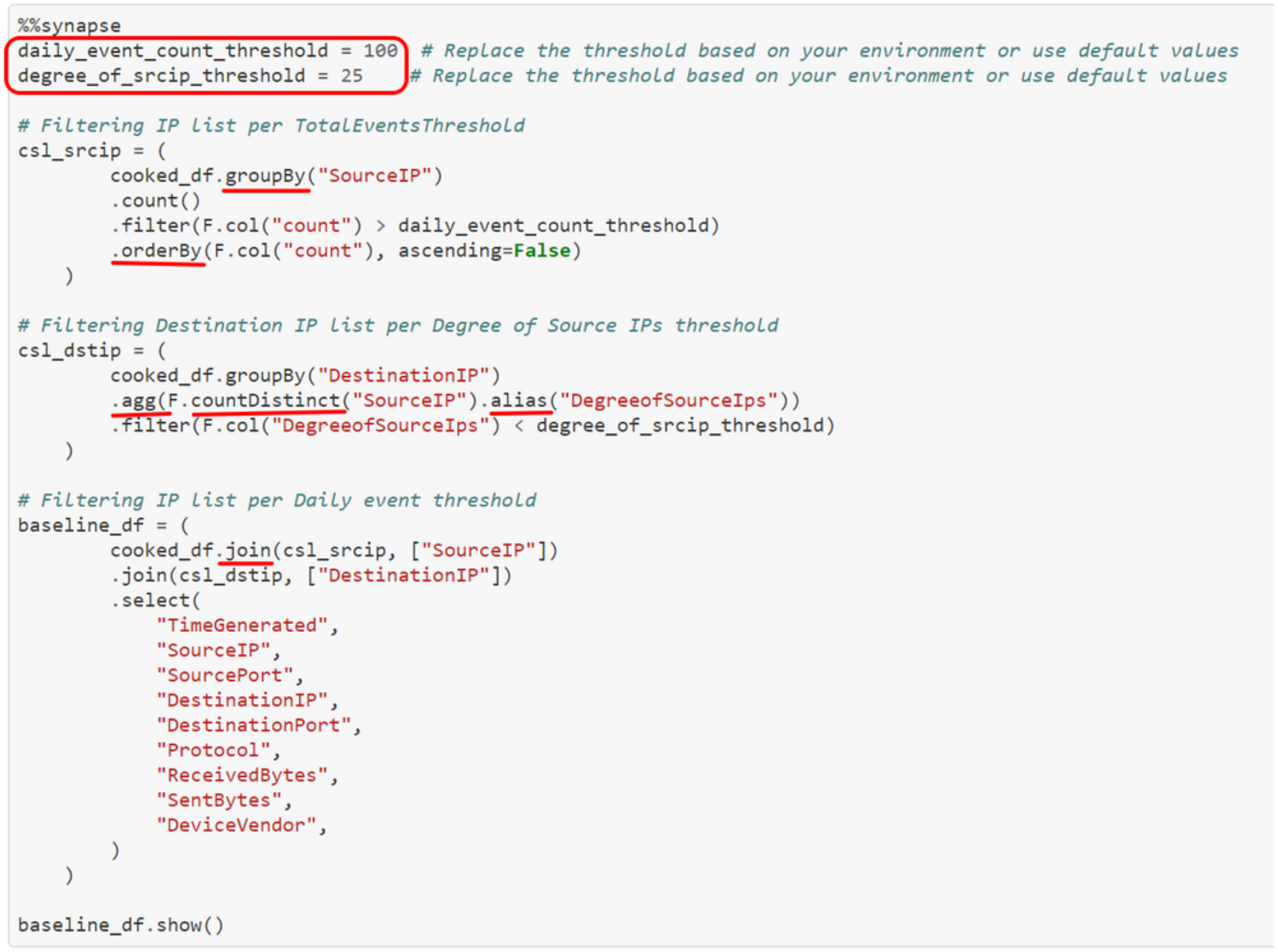
orderBy: Returns new dataframe sorted by specified column.

agg: Aggregate entire dataframe.

countDistinct: Return distinct count of column.

alias: Returns column aliased with new name.

join: Join another dataframe such as csl_srcip, csl_dstip.



```
%%synapse
daily_event_count_threshold = 100  # Replace the threshold based on your environment or use default values
degree_of_srcip_threshold = 25    # Replace the threshold based on your environment or use default values

# Filtering IP list per TotalEventsThreshold
csl_srcip = (
        cooked_df.groupBy("SourceIP")
        .count()
        .filter(F.col("count") > daily_event_count_threshold)
        .orderBy(F.col("count"), ascending=False)
    )

# Filtering Destination IP list per Degree of Source IPs threshold
csl_dstip = (
        cooked_df.groupBy("DestinationIP")
        .agg(F.countDistinct("SourceIP").alias("DegreeofSourceIps"))
        .filter(F.col("DegreeofSourceIps") < degree_of_srcip_threshold)
    )

# Filtering IP list per Daily event threshold
baseline_df = (
        cooked_df.join(csl_srcip, ["SourceIP"])
        .join(csl_dstip, ["DestinationIP"])
        .select(
            "TimeGenerated",
            "SourceIP",
            "SourcePort",
            "DestinationIP",
            "DestinationPort",
            "Protocol",
            "ReceivedBytes",
            "SentBytes",
            "DeviceVendor",
        )
    )

baseline_df.show()
```

Data Wrangling using Apache Spark

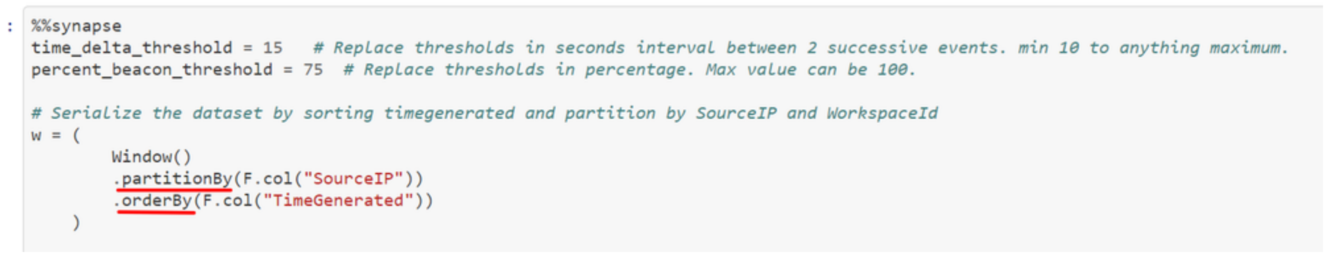
Once we have finished loading current data, historical data , prepare the dataset by filtering via regex and apply baselining to it, we can perform additional transformations on the PySpark dataframe.

- Sort the dataset by SourceIP.
- Calculate the time difference between first event and next event.
- Partition dataset per Source IP, Destination IP or Destination Port
- Group dataset into consecutive 3 rows to calculate the ***Timedeltalistcount*** which is aggregate of 3 respective timedelta counts
- Calculate ***percentagebeacon*** between ***TotalEventscount*** and ***Timedeltalistcount***
- Apply thresholds to further reduce false positives

Windowing

In the first step, we have sorted dataset and partitioned per SourceIP to calculate the time difference between the first and next row.

Window: Utility function for defining window in dataframe. In this case, we have created windows by partitioning by SourceIp first and later by multiple fields; DeviceVendor, SourceIP, DestinationIP, DestinationPort.



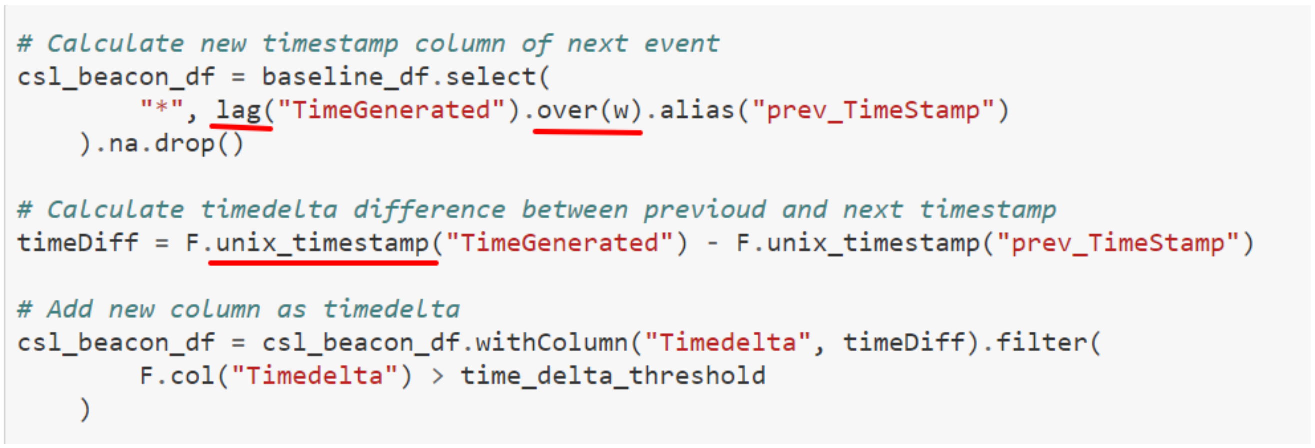
```
: %%synapse
time_delta_threshold = 15   # Replace thresholds in seconds interval between 2 successive events. min 10 to anything maximum.
percent_beacon_threshold = 75  # Replace thresholds in percentage. Max value can be 100.

# Serialize the dataset by sorting timegenerated and partition by SourceIP and WorkspaceId
w = (
        Window()
        .partitionBy(F.col("SourceIP"))
        .orderBy(F.col("TimeGenerated"))
    )
```

Time Delta Calculation

In this step, we have used lag to calculate time delta on the serialized and partitioned data obtained from previous step.

lag : Returns value that is offset rows before current row. In this case, we are calculating time delta between two consecutive events.

unix_timestamp: convert time string to Unix time stamp (in seconds) to return timedelta in seconds.

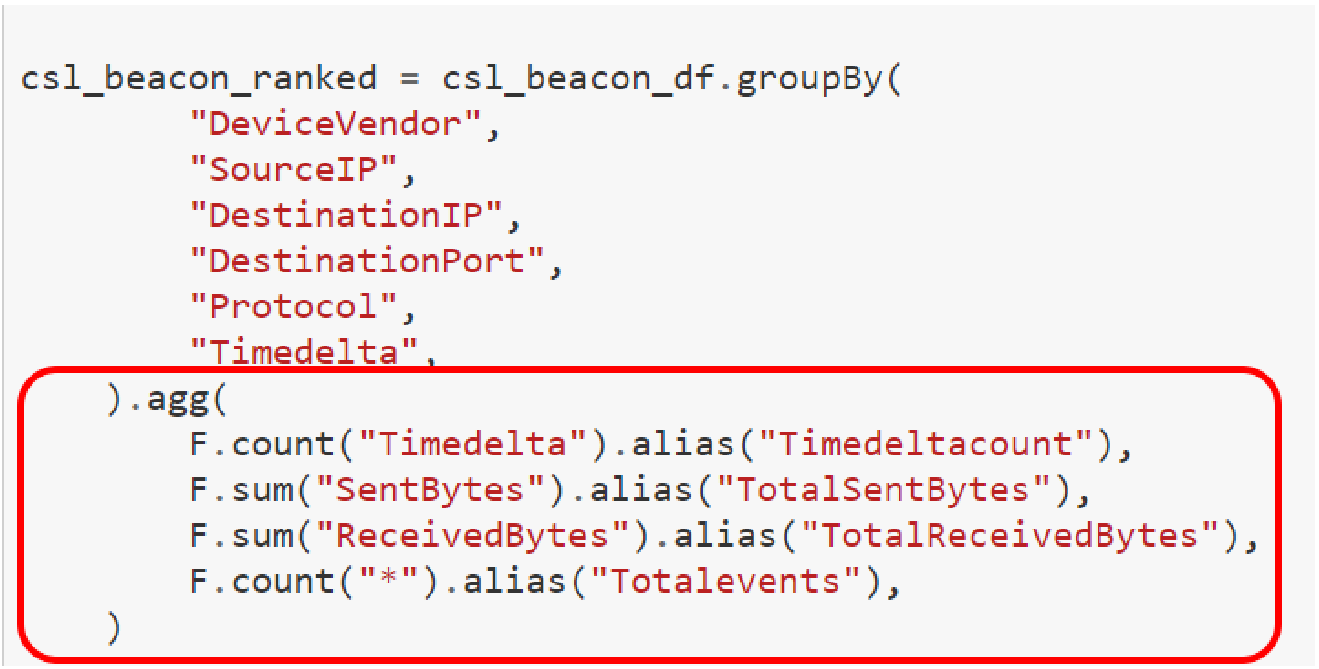


```
# Calculate new timestamp column of next event
csl_beacon_df = baseline_df.select(
        "*", lag("TimeGenerated").over(w).alias("prev_TimeStamp")
    ).na.drop()

# Calculate timedelta difference between previoud and next timestamp
timeDiff = F.unix_timestamp("TimeGenerated") - F.unix_timestamp("prev_TimeStamp")

# Add new column as timedelta
csl_beacon_df = csl_beacon_df.withColumn("Timedelta", timeDiff).filter(
        F.col("Timedelta") > time_delta_threshold
    )
```

We have also performed multiple aggregations to populate additional columns as shown in the screenshot.

sum : Computes sum for each numeric columns for each group.

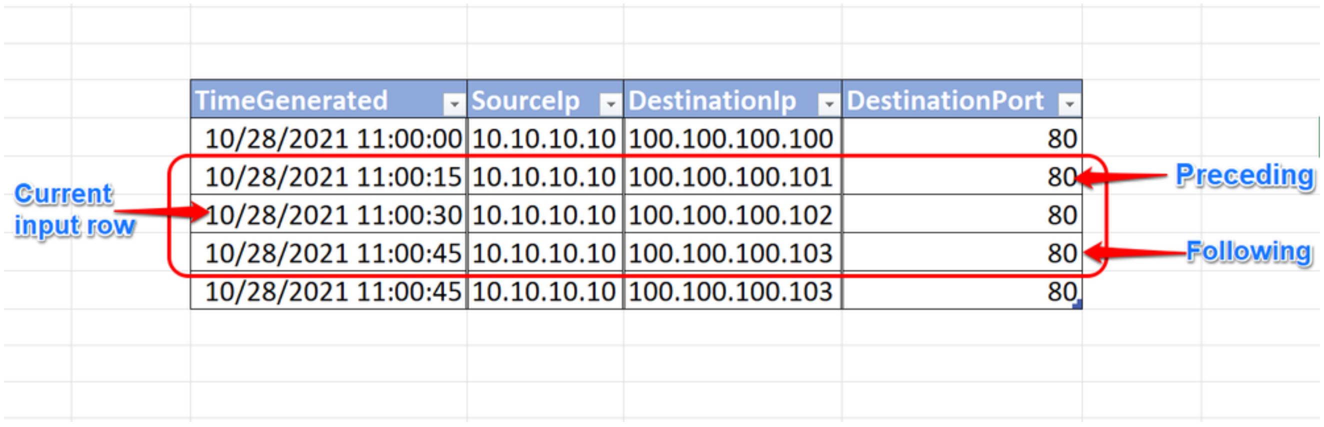
count: Returns number of rows in dataframe.


```
csl_beacon_ranked = csl_beacon_df.groupBy(
        "DeviceVendor",
        "SourceIP",
        "DestinationIP",
        "DestinationPort",
        "Protocol",
        "Timedelta",
    ).agg(
        F.count("Timedelta").alias("Timedeltacount"),
        F.sum("SentBytes").alias("TotalSentBytes"),
        F.sum("ReceivedBytes").alias("TotalReceivedBytes"),
        F.count("*").alias("Totalevents"),
    )
```

Repartition and Calculate Percentage Beaconing

Now, we can create a window specification with frame boundaries of 3 rows by partitioning based on DeviceVendor, SourceIP, DestinationIP, DestinationPort and order it by SourceIP in . This step will group dataset into consecutive 3 rows to calculate the Timedeltalistcount which is an aggregate of 3 respective timedelta counts.

Rowsbetween : Crates window spec with frame boundaries defined from start (inclusive) to end (inclusive).



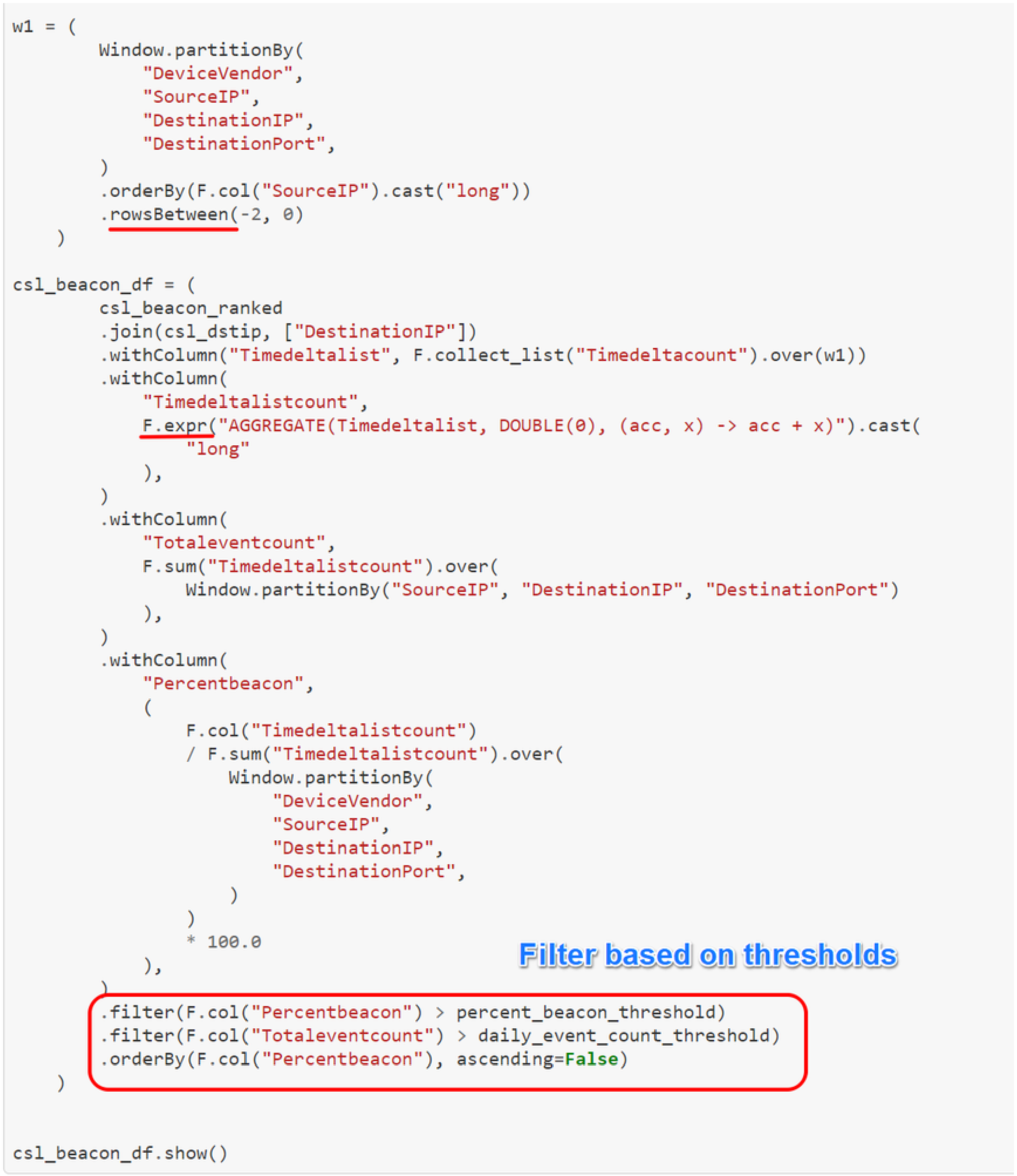
TimeGenerated	SourceIp	DestinationIp	DestinationPort
10/28/2021 11:00:00	10.10.10.10	100.100.100.100	80
10/28/2021 11:00:15	10.10.10.10	100.100.100.101	80
10/28/2021 11:00:30	10.10.10.10	100.100.100.102	80
10/28/2021 11:00:45	10.10.10.10	100.100.100.103	80
10/28/2021 11:00:45	10.10.10.10	100.100.100.103	80

Expr : Parses expression string into column. In this case we are calculating sum values of array integer.

Aggregate : aggregate(expre, start, merge,finish) – applies binary operator to an initial state and all elements of array and reduces this to a single state. In this case, we are aggregating all the timedeltalist and calculating respective sum of each.

We have also calculated PercentageBeacon which will be =

Aggregated count per time delta or list of time delta/ Total Events for Source-DestinationIP-Port * 100.

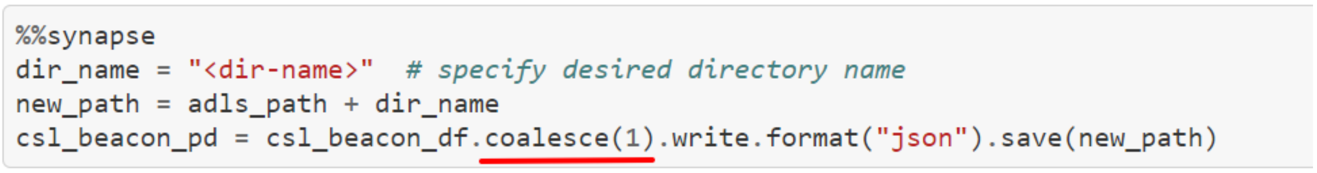


```
w1 = (
        Window.partitionBy(
            "DeviceVendor",
            "SourceIP",
            "DestinationIP",
            "DestinationPort",
        )
        .orderBy(F.col("SourceIP").cast("long"))
        .rowsBetween(-2, 0)
    )

csl_beacon_df = (
        csl_beacon_ranked
        .join(csl_dstip, ["DestinationIP"])
        .withColumn("Timedeltalist", F.collect_list("Timedeltacount").over(w1))
        .withColumn(
            "Timedeltalistcount",
            F.expr("AGGREGATE(Timedeltalist, DOUBLE(0), (acc, x) -> acc + x)").cast(
                "long"
            ),
        )
        .withColumn(
            "Totaleventcount",
            F.sum("Timedeltalistcount").over(
                Window.partitionBy("SourceIP", "DestinationIP", "DestinationPort")
            ),
        )
        .withColumn(
            "Percentbeacon",
            (
                F.col("Timedeltalistcount")
                / F.sum("Timedeltalistcount").over(
                    Window.partitionBy(
                        "DeviceVendor",
                        "SourceIP",
                        "DestinationIP",
                        "DestinationPort",
                    )
                )
                * 100.0
            ),
        )
        .filter(F.col("Percentbeacon") > percent_beacon_threshold)
        .filter(F.col("Totaleventcount") > daily_event_count_threshold)
        .orderBy(F.col("Percentbeacon"), ascending=False)
    )


csl_beacon_df.show()
```

Export Results from ADLS

Now you can export potential beaconing results retrieved from previous step and export it locally to be used outside the Azure Synapse session. To do, we have used SPARK API coalesce.

Coalesce: You can specify number of partitions (e.g. 1) to output dataframe results into single json file and write back to ADLS in the specified directory.

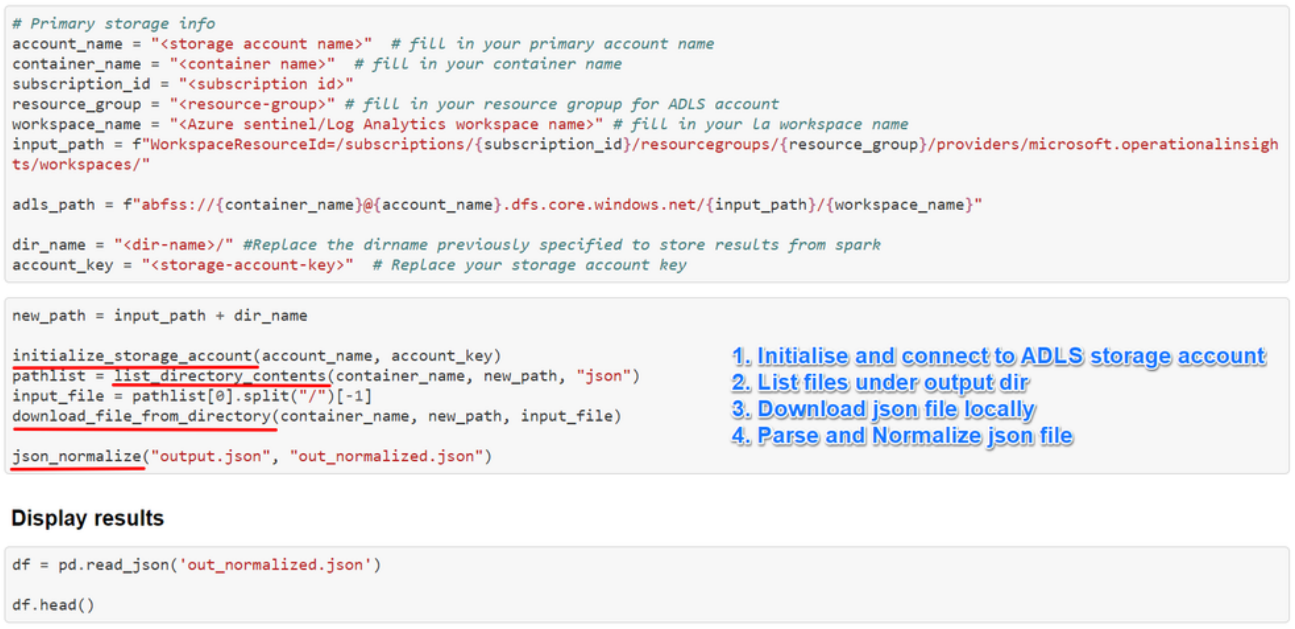


```
%%synapse
dir_name = "<dir-name>"  # specify desired directory name
new_path = adls_path + dir_name
csl_beacon_pd = csl_beacon_df.coalesce(1).write.format("json").save(new_path)
```

You can then stop the session by executing %synapse stop.

Once you are outside of synapse session, you can then download the output from ADLS locally and perform various data analysis, enrichment operations.

Specify the input parameters for ADLS account again since it is outside synapse session, it won't recognize the variables declared inside synapse session.



```
# Primary storage info
account_name = "<storage account name>"  # fill in your primary account name
container_name = "<container name>"  # fill in your container name
subscription_id = "<subscription id>"
resource_group = "<resource-group>" # fill in your resource gropup for ADLS account
workspace_name = "<Azure sentinel/Log Analytics workspace name>" # fill in your la workspace name
input_path = f"WorkspaceResourceId=/subscriptions/{subscription_id}/resourcegroups/{resource_group}/providers/microsoft.operationalinsigh
ts/workspaces/"

adls_path = f"abfss://{container_name}@{account_name}.dfs.core.windows.net/{input_path}/{workspace_name}"

dir_name = "<dir-name>/" #Replace the dirname previously specified to store results from spark
account_key = "<storage-account-key>"  # Replace your storage account key
```

```
new_path = input_path + dir_name

initialize_storage_account(account_name, account_key)
pathlist = list_directory_contents(container_name, new_path, "json")
input_file = pathlist[0].split("/")[-1]
download_file_from_directory(container_name, new_path, input_file)

json_normalize("output.json", "out_normalized.json")
```

1. Initialise and connect to ADLS storage account
2. List files under output dir
3. Download json file locally
4. Parse and Normalize json file

Display results

```
df = pd.read_json('out_normalized.json')

df.head()
```

Enriching entities with MSTICPy for investigation

In order to investigate the beaconing results , we can further automate the entity enrichment tasks such as GeoIP lookup, Whois lookup and ThreatIntel lookups using native features of MSTICPy library.

You can also visualize results onto geographical map using FoliumMap visualization of MSTICPy.

Please refer MSTICPy data enrichment and visualization for detailed information along with example notebooks. Once you have enriched results, you can then send those results back to Sentinel as bookmark by using MSTICPy Microsoft Sentinel APIs.

Conclusion

The notebook implemented use case outlined previously in the blog using Apache spark , applied baselining methods on historical datasets to further reduce datasets and also enriched entities with useful information using MSTICPy features to automate common investigation steps for analysts. You can further take these results into Microsoft Sentinel to either combine with other datasets or alerts to increase fidelity or create incidents to investigate the reasoning behind the patterns observed. In the second part of the blog, we will take simulated datasets to test this notebook and analyze the results.

References

- Detect Network beaconing via Intra-Request time delta patterns in Microsoft Sentinel - Microsoft Tec...
- Large Scale analytics with Azure Synapse and Microsoft Sentinel Notebooks
- Guided Hunting Notebook : https://aka.ms/Beacon_Synapse_Notebook
- Apache PySpark SQL : https://spark.apache.org/docs/3.1.1/api/python/reference/pyspark.sql.html
- Introducing Window Functions in Spark SQL - The Databricks Blog
- https://stackoverflow.com/questions/47839077/pyspark-best-way-to-sum-values-in-column-of-type-arrayi...
- https://msticpy.readthedocs.io/en/latest/data_acquisition/GeoIPLookups.html
- https://msticpy.readthedocs.io/en/latest/data_acquisition/TIProviders.html
- https://msticpy.readthedocs.io/en/latest/visualization/FoliumMap.html