Lab: Performing Real-Time Intelligence using Microsoft Fabric

Objective

- Create **KQL database** in your Fabric enabled Power BI workspace
- Run powerful KQL queries to explore the data via **KQL Queryset**
- Visualize data in Fabric **Real Time Dashboards**

Exercise 1: Database Creation, Data Ingestion and Exploration

Exercise 2: Ingest data from Azure Storage Account

Exercise 3: Starting with the basics of KQL

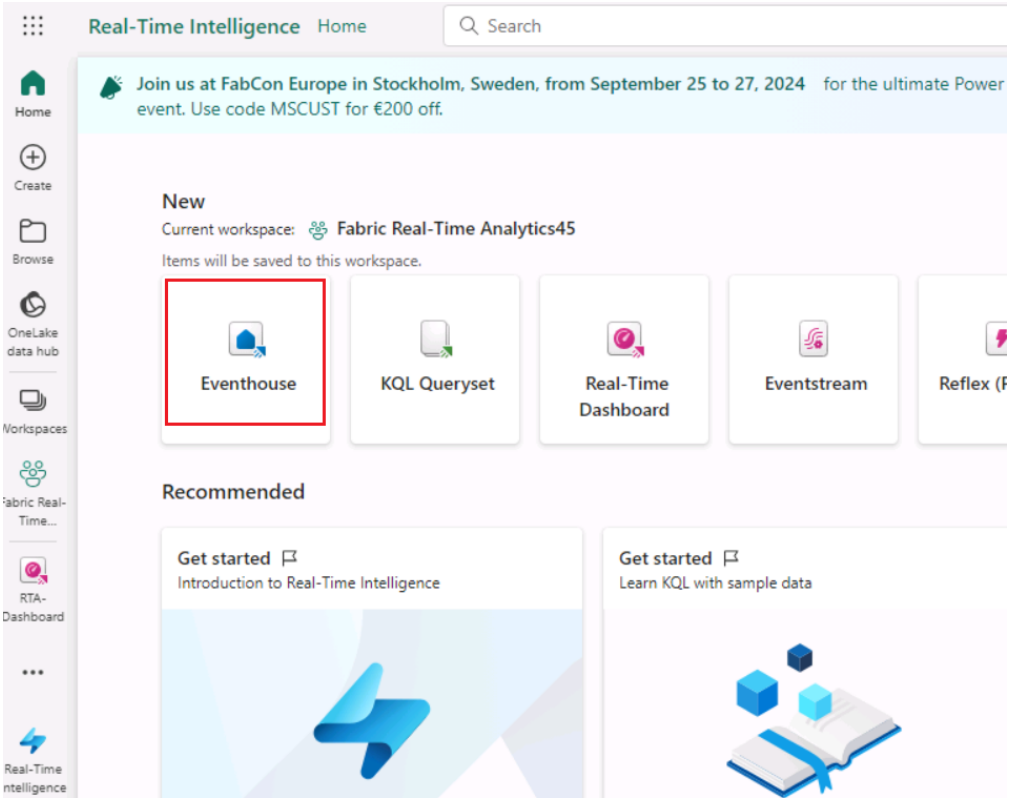
Exercise 4: Explore and Transform Data

Exercise 5: Eventstreams

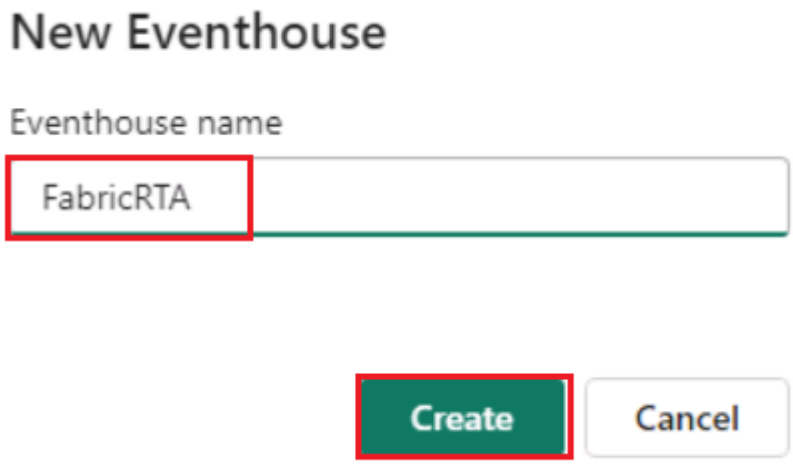
Exercise 1: Database Creation, Data Ingestion and Exploration

Task 1: Create a KQL Database

Kusto query language (KQL) is used to query static or streaming data in a table that is defined in a KQL database. To analyze the sales data, you must create a table in a KQL database and ingest the data from the file.



In the **New Eventhouse** dialog box, enter the **Eventhouse name** as **FabricRTA** (or new database with a name of your choice) and click on **Create** button.

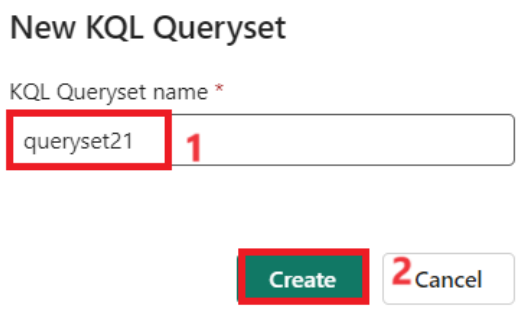


Task 2: Create a KQL queryset

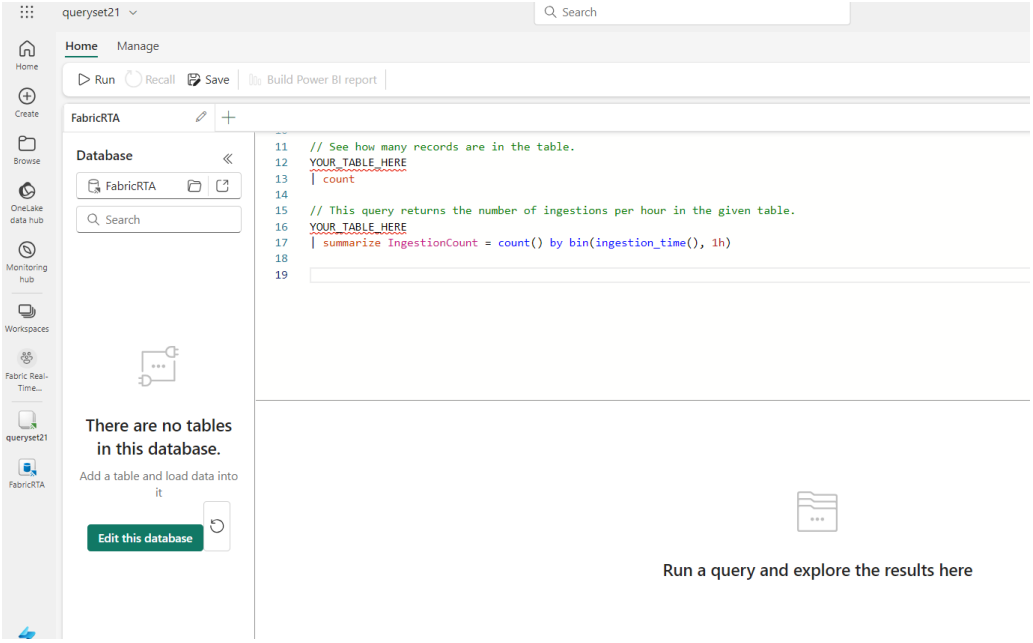
The KQL Queryset exists within the context of a workspace. A new KQL queryset is always associated with the workspace you're using when you create it.

1. Select **Your** *workspace* in the left navigation pane.

2. Select the **New** drop-down, and from there select **KQL Queryset**
3. In the **New KQL Queryset** dialog box, enter the **KQL Query name** as querysetXX (or new database with a name of your choice) and click on **Create** button.

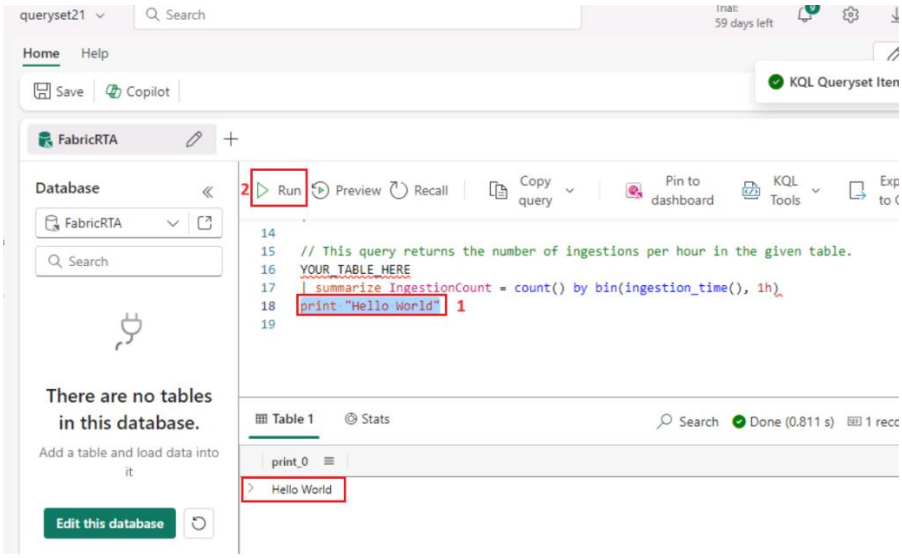


4. In **OneLake data hub** pane ,select **FabricRTA** KQL Database and click on **Select** button.

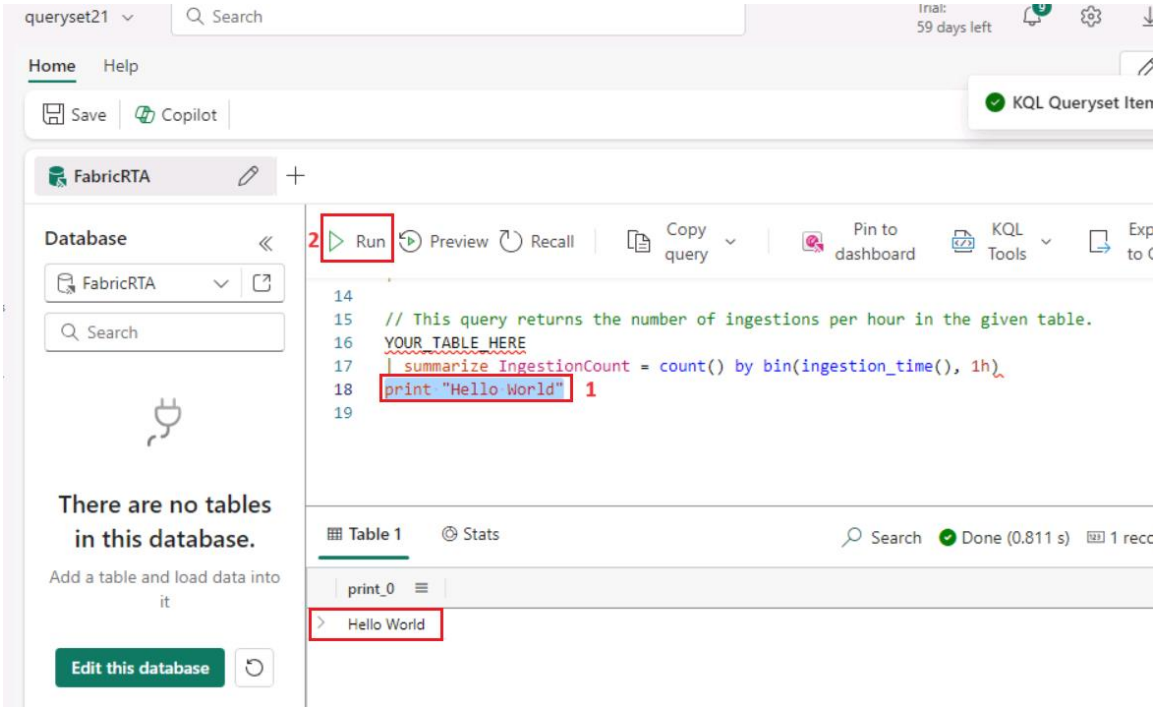


5. Now you can write a simple KQL query:

   print "Hello World"



6. Highlight the line with the code and hit the **Run** button. The query will be executed and its result can be seen in the result grid at the bottom of the page.


queryset21
Search
Trial:
59 days left
Home
Help
Save
Copilot
KQL Queryset Iten
FabricRTA
Database
FabricRTA
Search
There are no tables in this database.
Add a table and load data into it
Edit this database
2
Run
Preview
Recall
Copy query
Pin to dashboard
KQL Tools
14
15 // This query returns the number of ingestions per hour in the given table.
16 YOUR_TABLE_HERE
17 | summarize IngestionCount = count() by bin(ingestion_time(), 1h)
18 print "Hello World"
1
19
Table 1
Stats
Search
Done (0.811 s)
1 recc
print_0
Hello World

Exercise 2: Ingest data from Azure Storage Account

Data ingestion to KQL Database is the process used to load data records from one or more sources into a table. Once ingested, the data becomes available for query.

KQL Database supports several ingestion methods, including Eventstream, Fabric Pipeline, and Fabric Dataflow. Also available with Azure Data Factory and Event Hubs.

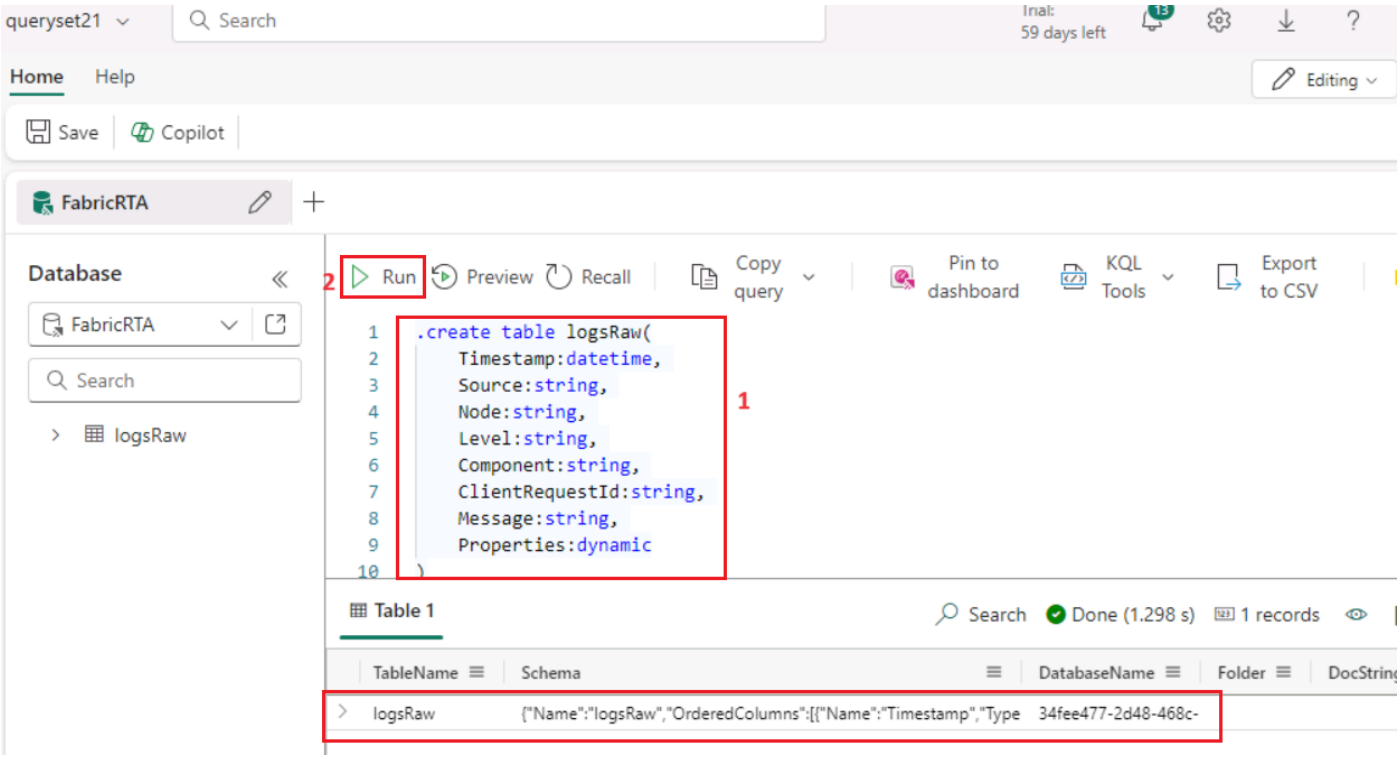
Task 1: Create the raw table - logsRaw

Ingest data using one-click ingestion from Azure Blob Storage to your KQL Database.

1. In KQL queryset pane, replace all the code in the **cell** with the following code and click on ▷ **Run** button to create a table

Copy

```
.create table logsRaw(
    Timestamp:datetime,
    Source:string,
    Node:string,
    Level:string,
    Component:string,
    ClientRequestId:string,
    Message:string,
    Properties:dynamic
)
```

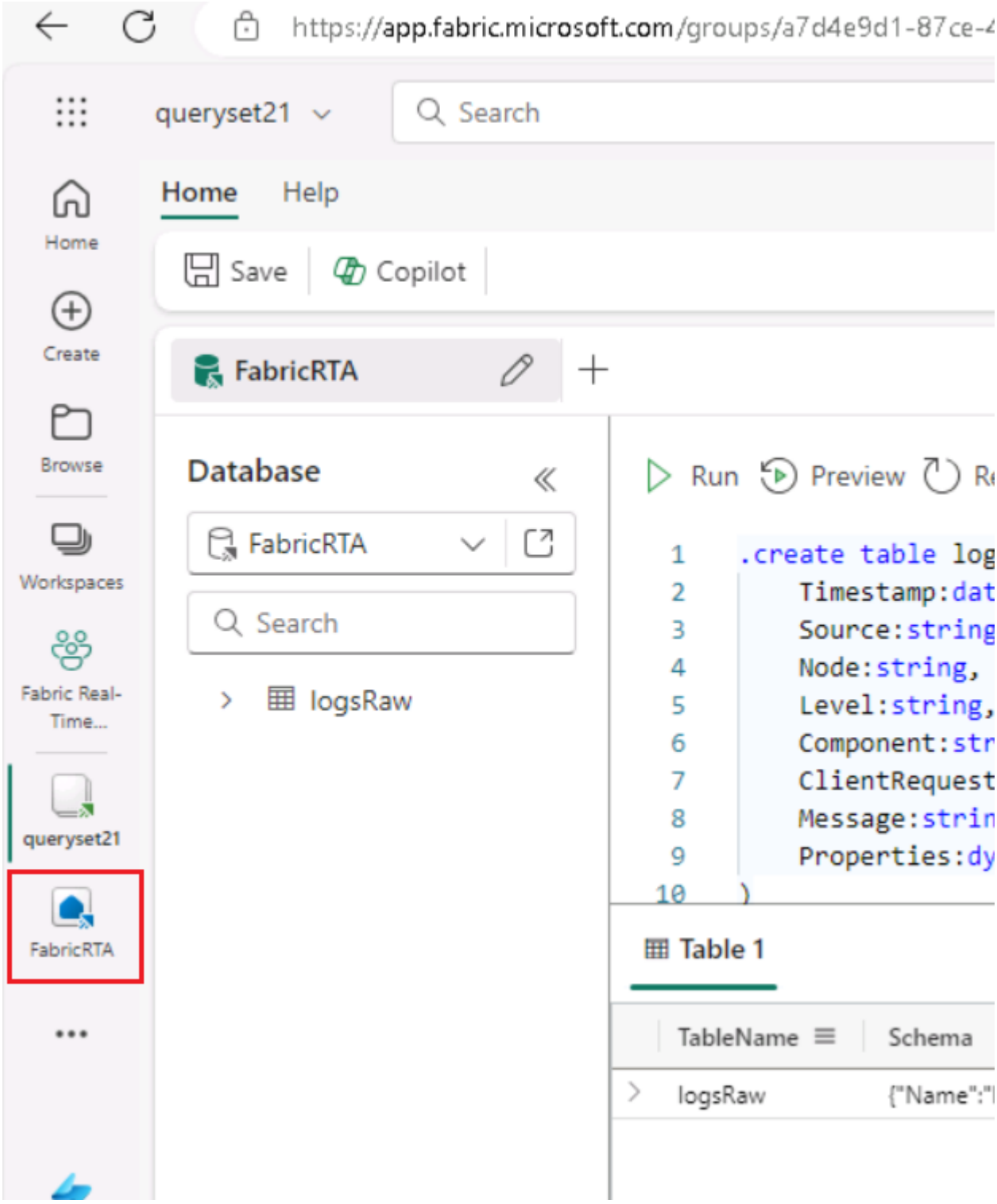


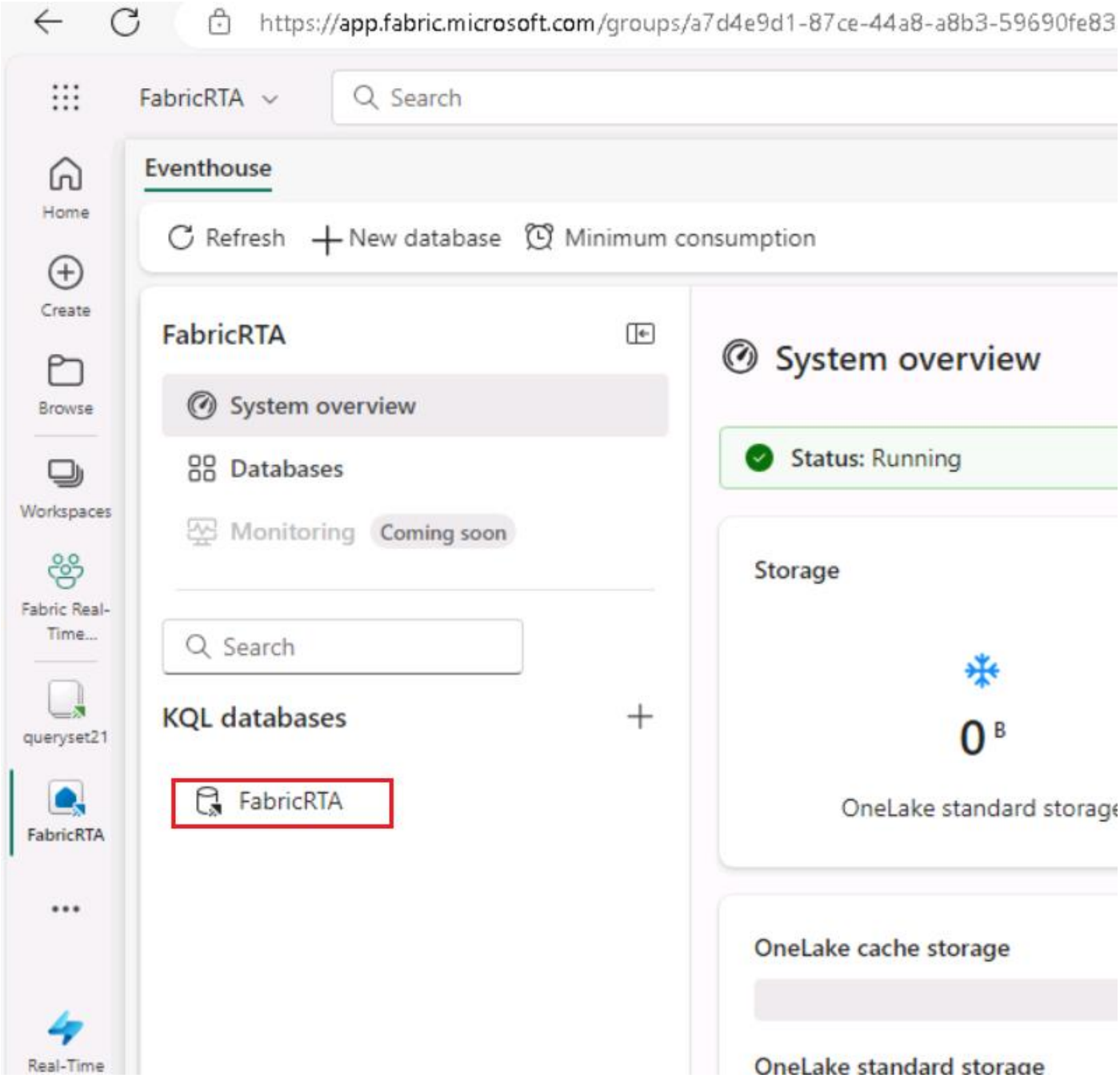
2. Click on refresh the page to see the new table on the left.

Task 2: Use the "One-click" User Interface to ingest data from Azure blob storage

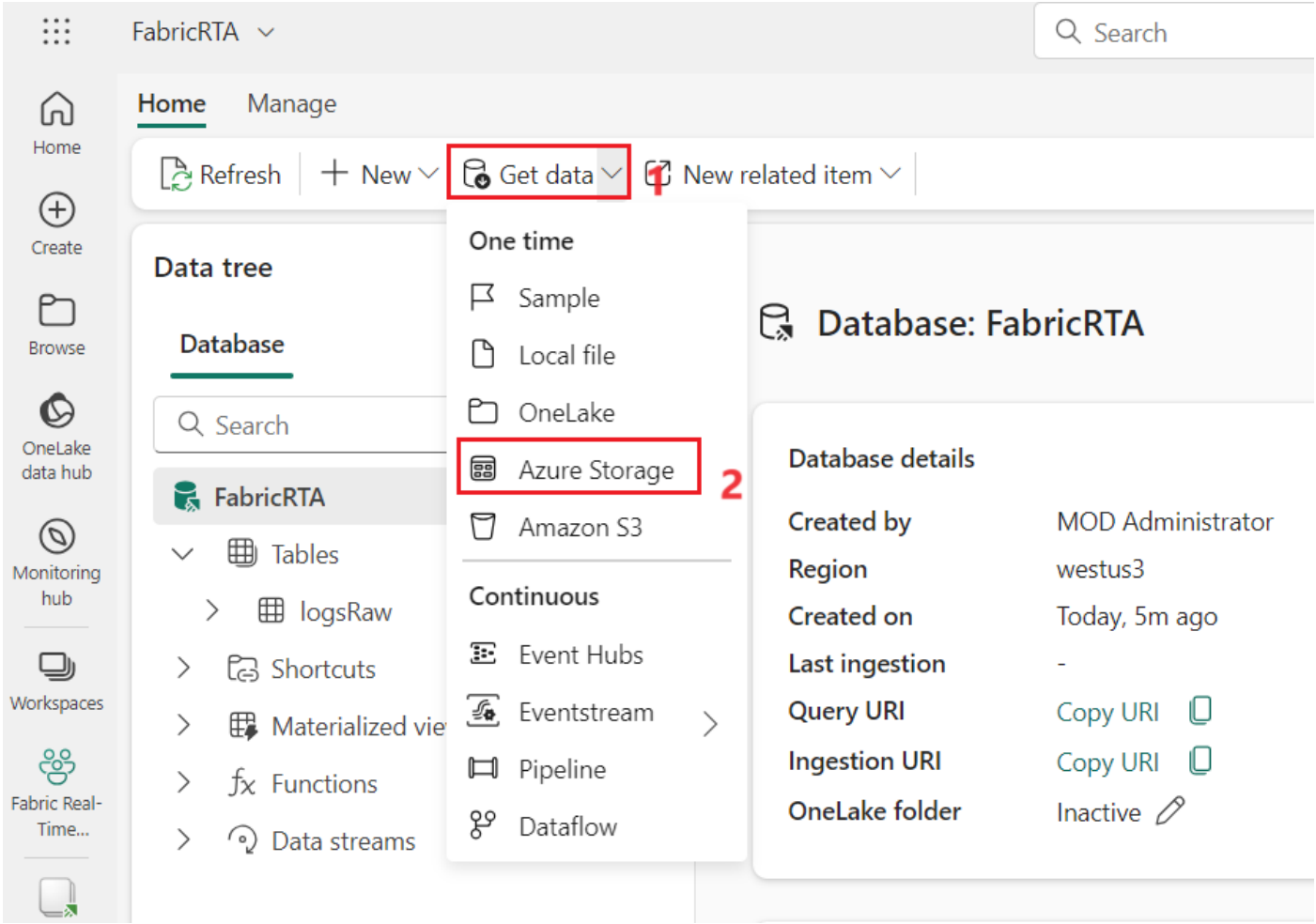
You need to analyze the system logs for Contoso, which are stored in Azure blob storage.

1. Select **FabricRTA** *database* in the left navigation pane.

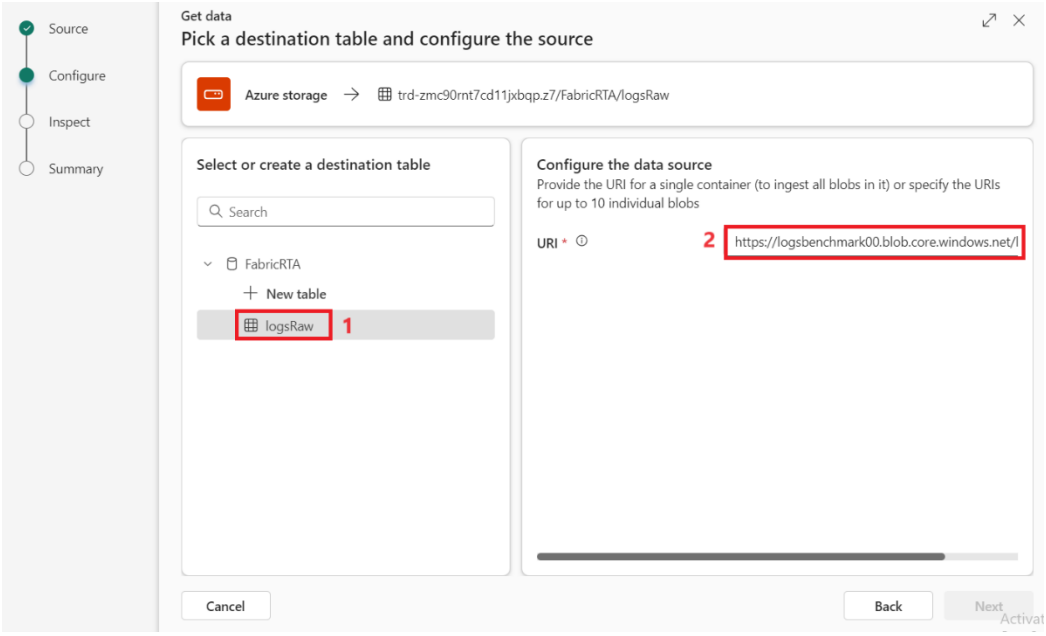




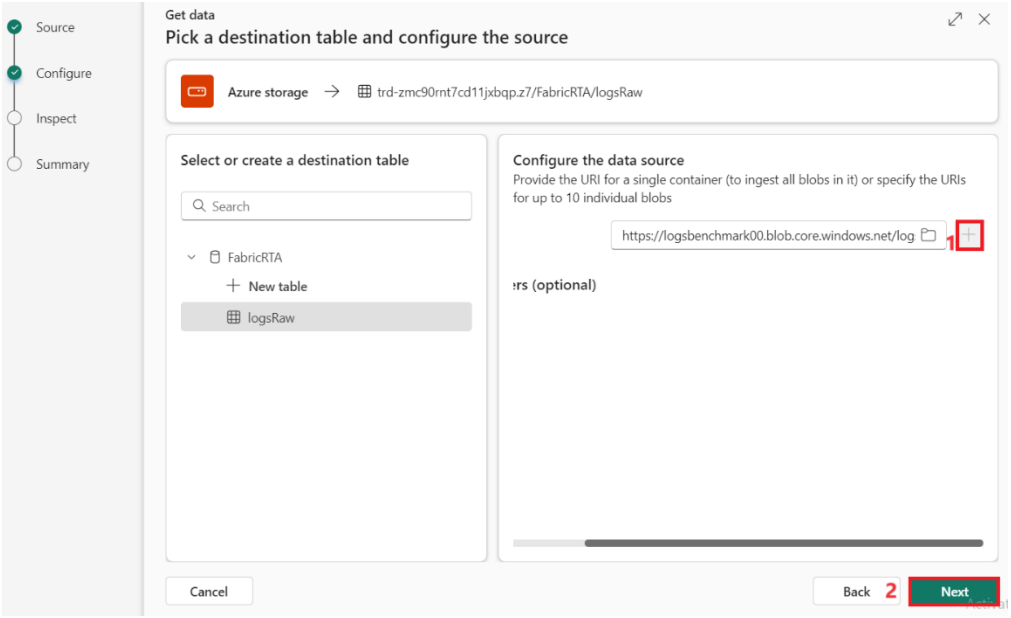
2. In **FabricRTA** database, under the **Home** tab, navigate and click on **Get data** in the command bar, then select **Azure Storage**.



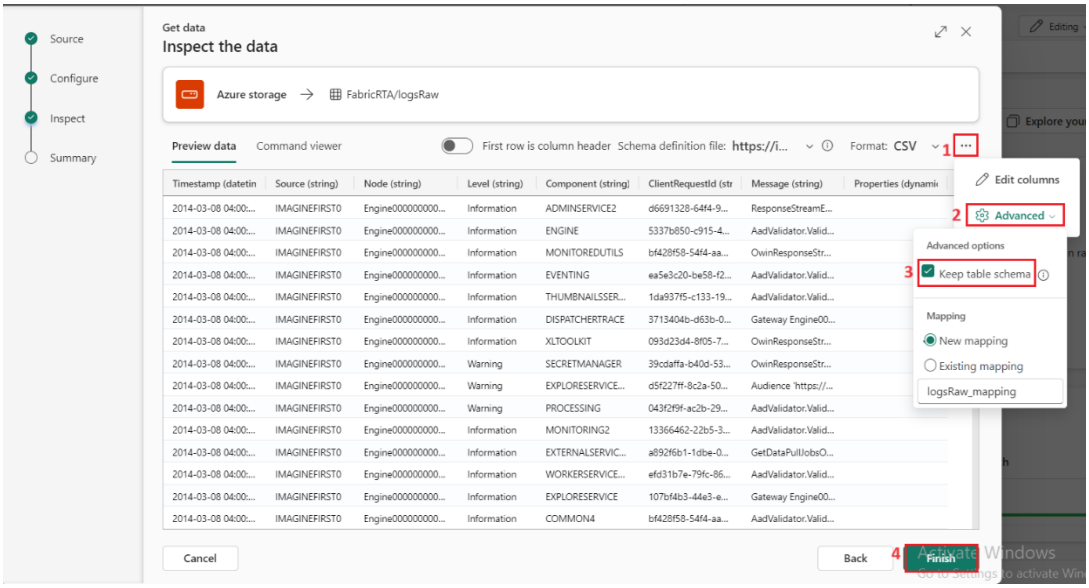
3. Then use the wizard to import the data into a new table by selecting the following options:
4. In the **Get data** tab, select the existing table as **logsRaw**. Under the Configure the data source tab enter the **URI :** https://rtainaday.blob.core.windows.net/logsbenchmark-onegb/2014?sp=rl&st=2025-03-12T19:25:59Z&se=2099-03-13T03:25:59Z&spr=https&sv=2022-11-02&sr=c&sig=XxQnOYzmqarwNi8xhRF7zbRMuY8T0WXmz5CVCJMiW0M%3D



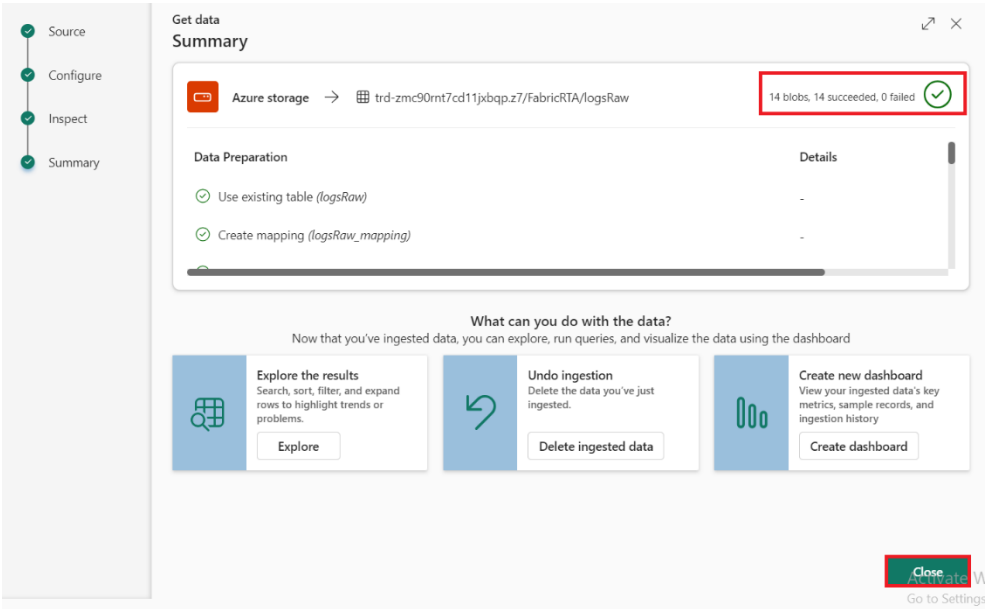
5. In the **Get data** tab, click on the **+** and click on **Next** button.



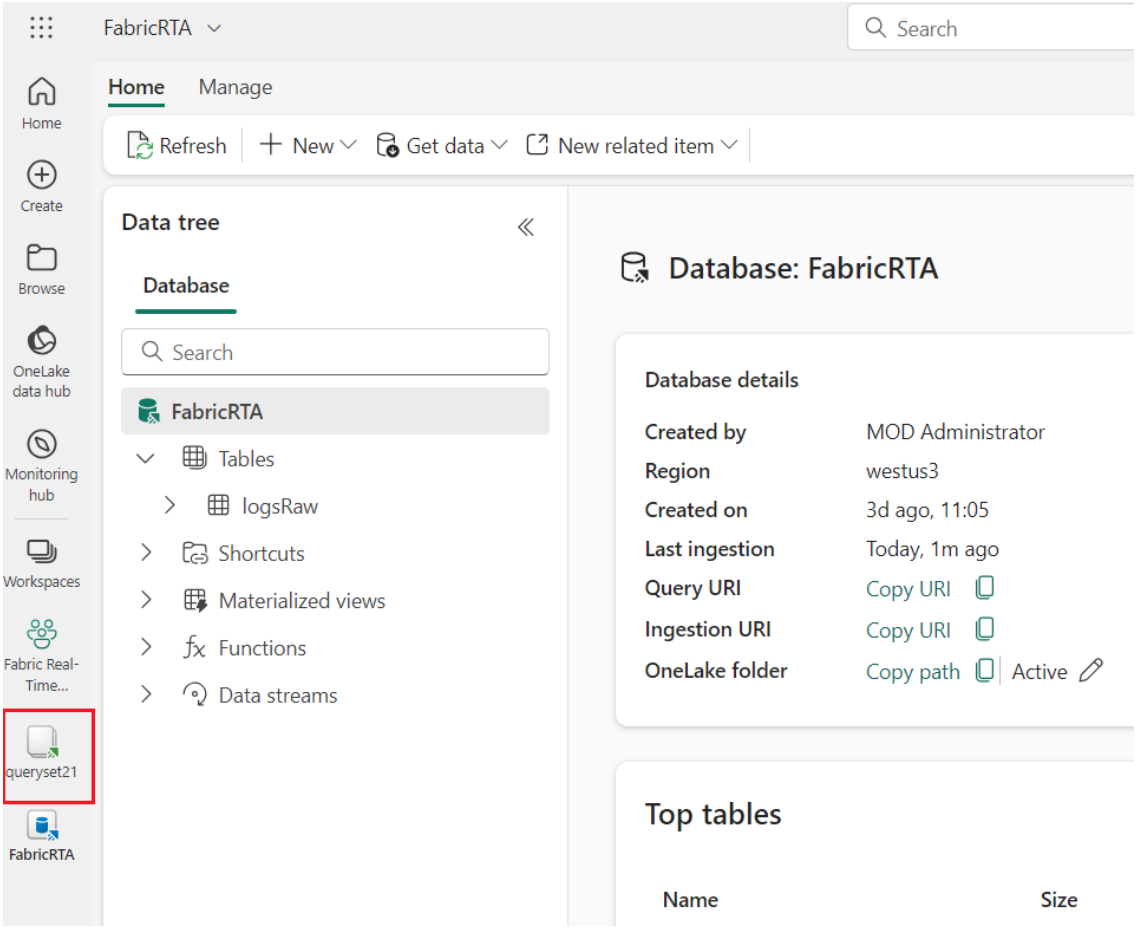
6. In Get data tab under the Inspect the data click on the horizontal **ellipses (…)** beside Edit columns, select the **Advanced** and check the **Keep table schema.** Click on the **Finish** button**.**



7. Wait for the Data ingestion to be completed, and click **Close**.

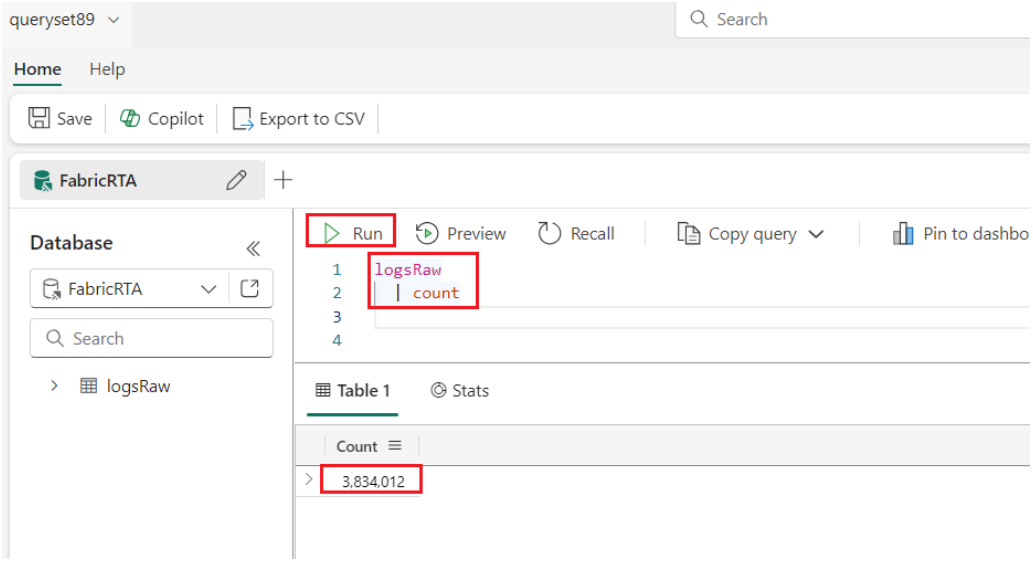


8. Select **querysetXX** KQL queryset in the left navigation pane.



9. Replace all the code in the **cell** with the following code and click on ▷ **Run** button

```
logsRaw
| count
```



The logsRaw table should have 3,834,012 records.

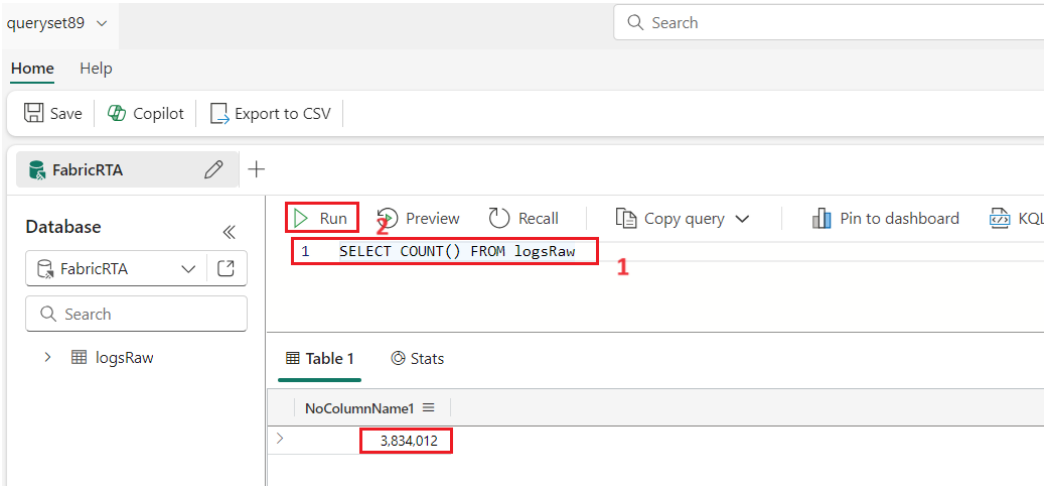
Exercise 3: Starting with the basics of KQL

A *Kusto query* is a read-only request to process data and return results. The request is stated in plain text that's easy to read, author, and automate. A Kusto query has one or more query statements and returns data in a tabular or graph format.

Task 1: Journey from SQL to KQL!

1. For all the SQL pros out there, KQL Database allows a subset of TSQL queries. Try running the following SQL query in web UI. Replace all the code in the **cell** with the following code and click on ▷ **Run** button.

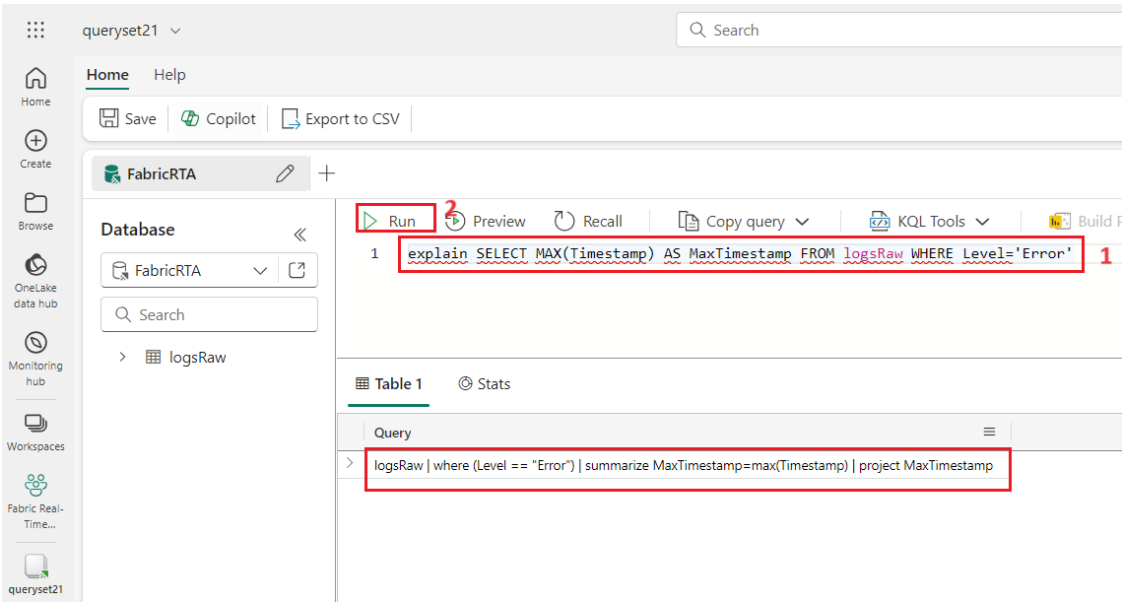
SELECT COUNT() FROM logsRaw



2. The primary language to interact with Kusto is KQL (Kusto Query Language). To make the transition and learning experience easier, you can use the explain operator to translate SQL queries to KQL.

3. Replace all the code in the **cell** with the following code and click on ▷ **Run** button.

explain SELECT MAX(Timestamp) AS MaxTimestamp FROM logsRaw WHERE Level='Error'



Output of the above query will be a corresponding KQL query

logsRaw

| where (Level == "Error")

| summarize MaxTimestamp=max(Timestamp)

| project MaxTimestamp

Task 2: Basic KQL queries - explore the data

In this task, you will see some KQL examples. For this task, we will use the table logsRaw, which has data we loaded in previous challenge from storage account.

1. Execute the queries and view the results. KQL queries can be used to filter data and return specific information. Now, you'll learn how to choose specific rows of data. The **where** operator filters results that satisfy a certain condition.

2. Replace all the code in the **cell** with the following code and click on ▷ **Run** button.

```
logsRaw
| where Level=="Error"
| take 10
```

3. Find out how many records are in the table. Replace all the code in the **cell** with the following code and click on ▷ **Run** button.

```
logsRaw
| summarize count()
```

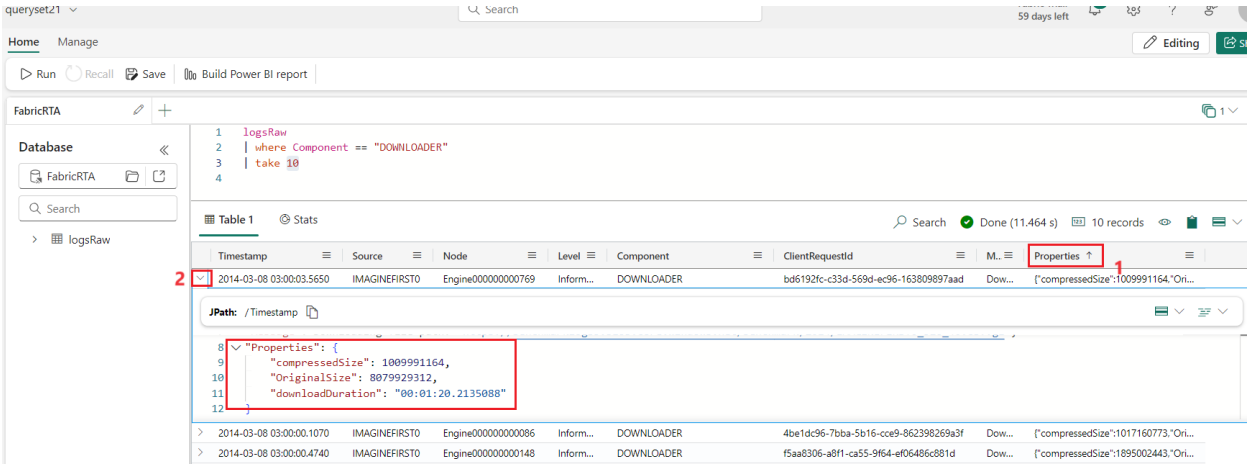
4. Find out the minimum and maximum Timestamp. Replace all the code in the **cell** with the following code and click on ▷ **Run** button.

```
logsRaw
| summarize min(Timestamp), max(Timestamp)
```

5. Our dataset has trace records written by Contoso's DOWNLOADER program, which downloads files from blob storage as part of its business operations. Replace all the code in the **cell** with the following code and click on ▷ **Run** button.

```
logsRaw
| where Component == "DOWNLOADER"
| take 10
```

6. Select the **Properties** column is dynamic. The dynamic data type is special in that it can take on any value of other data types, as well as arrays and property bags (dictionaries).



7. The dynamic type is extremely beneficial when it comes to storing JSON data, since KQL makes it simple to access fields in JSON and treat them like an independent column: just use either the dot notation (dict.key) or the bracket notation (dict["key"]).

8. The extend operator adds a new calculated column to the result set, during query time. This allows for the creation of new standalone columns to the result set, from the JSON data in dynamic columns. Replace all the code in the **cell** with the following code and click on ▷ **Run** button.

```
logsRaw
| where Component == "DOWNLOADER"
| take 100
| extend originalSize=Properties.OriginalSize, compressedSize=Properties.compressedSize
```

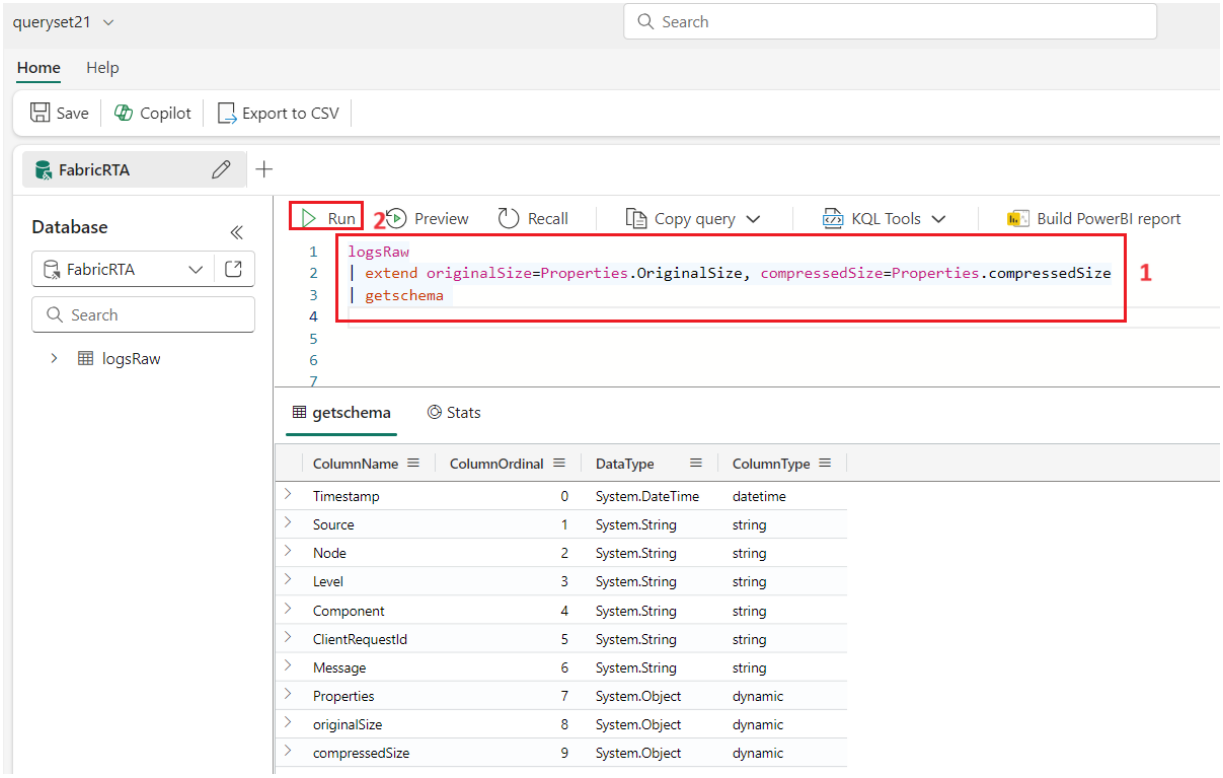
Task 3: Explore the table and columns

After subscripting a dynamic object, it is necessary to cast (convert) the value to a simple type in order to utilize them (for example, if you want to summarize the sizes of all the OriginalSize, you should convert the dynamic type to a numeric type, like long).

1. This below query to get the table that is shown in the image below (we want to convert the OriginalSize and CompressedSize columns to long)

2. Replace all the code in the **cell** with the following code and click on ▷ **Run cell** button and review the output.

```
logsRaw
| extend originalSize=Properties.OriginalSize, compressedSize=Properties.compressedSize
| getschema
```

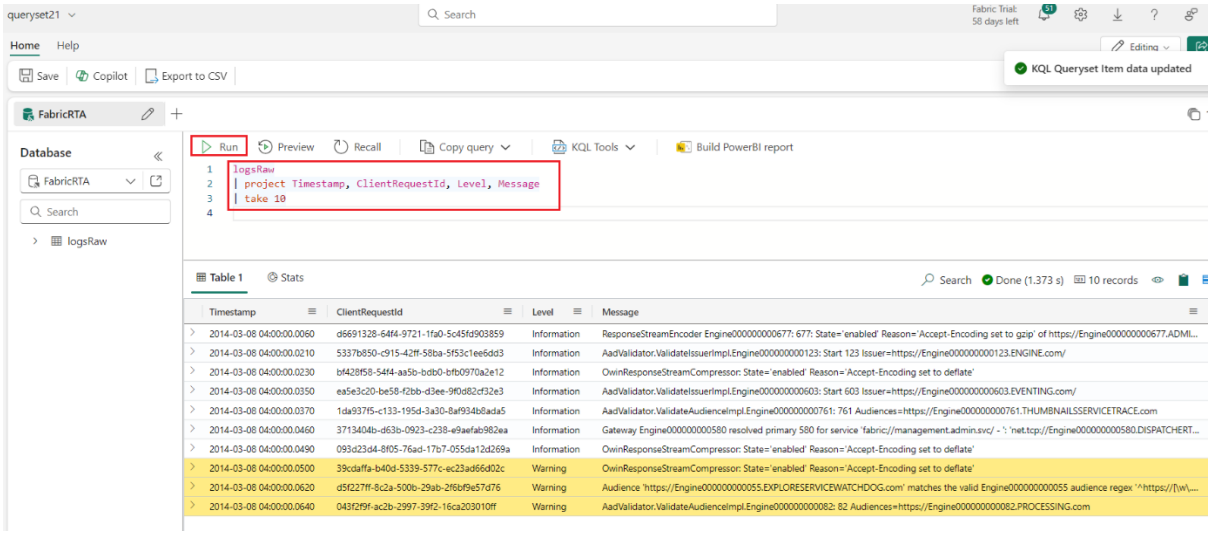


Task 4: Keep the columns of your interest

You are investigating an incident and wish to review only several columns of the dataset.

1. A query to get only specific desired columns: Timestamp, ClientRequestId, Level, Message. Take arbitrary 10 records.
2. Replace all the code in the **cell** with the following code and click on ▷ **Run cell** button and review the output.

```
logsRaw
| project Timestamp, ClientRequestId, Level, Message
| take 10
```



Task 5: Filter the output

You are investigating an incident that occurred within a specific time frame.

1. Write a query to get only specific desired columns: Timestamp, ClientRequestId, Level, Message. Take all the records between 2014-03-08 01:00 and 2014-03-08 10:00.
2. Replace all the code in the **cell** with the following code and click on ▷ **Run cell** button and review the output.

```
logsRaw
| where Timestamp between (datetime(2014-03-08 01:00).. datetime(2014-03-08 10:00))
| project Timestamp, ClientRequestId, Level, Message
| take 10
```

Task 6: Sorting the results

Your system generated an alert indicating a significant decrease in incoming data. You want to check the traces of the "INGESTOR_EXECUTER" [sic] component of the program.

1. Write a query that returns 20 sample records in which the Component column equals the word "INGESTOR_EXECUTER" [sic].
2. Once done, rewrite the query to take the top 1 records by the value of rowCount (for the "INGESTOR_EXECUTER" [sic] records).
3. Replace all the code in the **cell** with the following code and click on ▷ **Run cell** button and review the output.

```
logsRaw
| where Component == 'INGESTOR_EXECUTER'
| extend rowCount=toint(Properties.rowCount)
| where isnotempty(rowCount)
| sort by rowCount
| top 10 by rowCount desc
```

Task 7: Data profiling

1. As part of the incident investigation, you want to extract format and rowCount from INGESTOR_EXECUTER [sic] component. Rename the calculated fields to fileFormat and rowCount respectively. Also, Make Sure Timestamp, fileFormat and rowCount are the first 3 columns.
2. Replace all the code in the **cell** with the following code and click on ▷ **Run cell** button and review the output.

```
logsRaw
| where Component == 'INGESTOR_EXECUTER'
| extend rowCount=toint(Properties.rowCount), fileFormat=tostring(Properties.format)
| project Timestamp, fileFormat, rowCount, ClientRequestId, Component, Level, Message
| take 10
```

Task 8: Total number of records

The system comprises of several "components", but you don't know their names or how many records were generated by each.

1. Write a query to find out how many records were generated by each component. Use the Component column.
2. Replace all the code in the **cell** with the following code and click on ▷ **Run cell** button and review the output.

Copy

```
logsRaw
| summarize count() by Component
```

Task 9: Aggregations and string operations

You assume that the incident being investigated has a connection to the ingestion process run by Contoso's program.

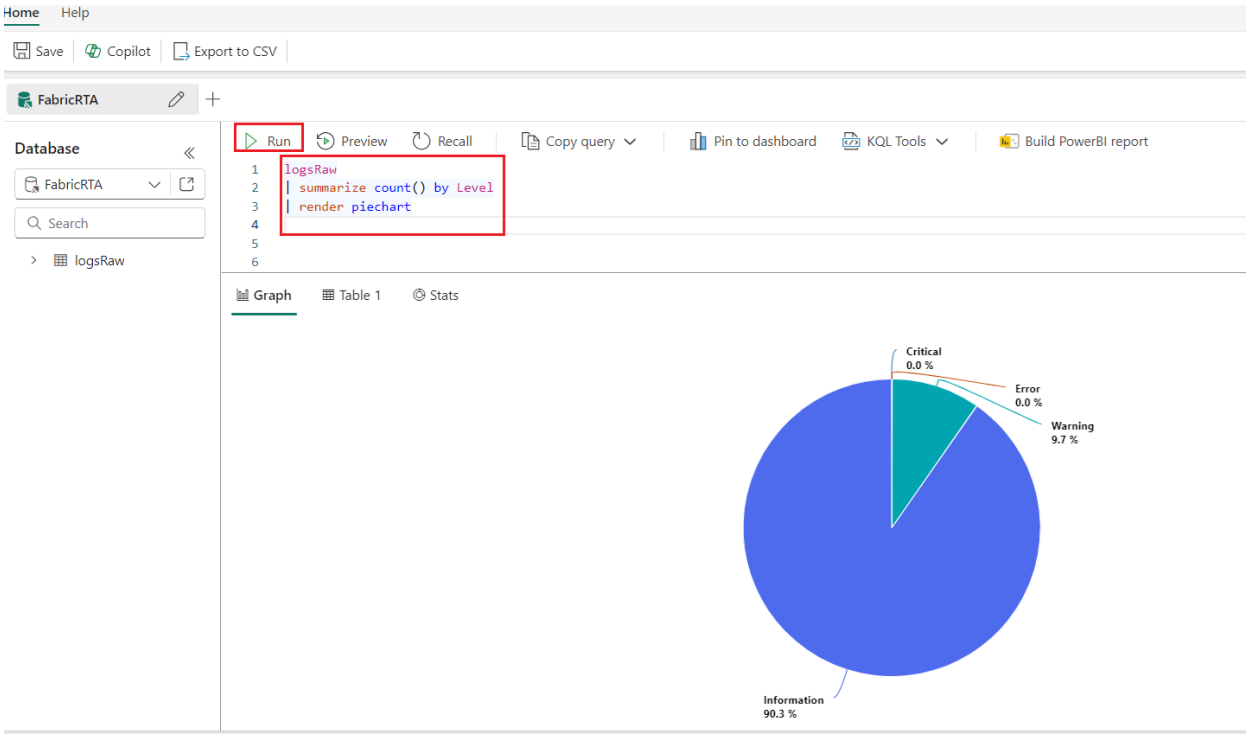
1. Write a query to find out how many records contain the string '**ingestion'** in the Message column. Aggregate the results by **Level.**
2. Replace all the code in the **cell** with the following code and click on ▷ **Run cell** button and review the output.

```
logsRaw
| where Message has "ingestion"
| summarize count() by Level
```

Task 10: Render a chart

1. Write a query to find out how many total records are present per Level (aggregated by Level) and render a piechart.
2. Replace all the code in the **cell** with the following code and click on ▷ **Run cell** button and review the output.

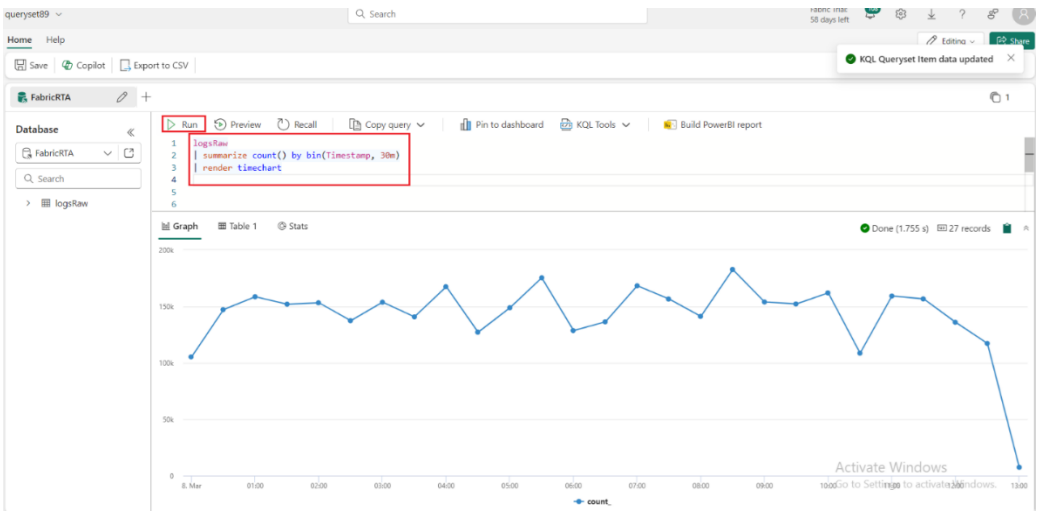
```
logsRaw
| summarize count() by Level
| render piechart
```



Task 11: Create bins and visualize time series

1. Write a query to show a timechart of the number of records in 30 minute bins (buckets). Each point on the timechart represent the number of logs in that bucket.
2. Replace all the code in the **cell** with the following code and click on ▷ **Run cell** button and review the output.

```
logsRaw
| summarize count() by bin(Timestamp, 30m)
| render timechart
```



Exercise 4: Explore and Transform Data

In this exercise we will explore 3 capabilities of Data Explorer

- **User-defined functions** are reusable KQL subqueries that can be defined as part of the query itself (ad-hoc functions), or persisted as part of the database metadata (stored functions - reusable KQL query, with the given name). Stored functions are invoked through a name, are provided with zero or more input arguments (which can be scalar or tabular), and produce a single value (which can be scalar or tabular) based on the function body.
- **Update Policy** is like an internal ETL. It can help you manipulate or enrich the data as it gets ingested into the source table (e.g. extracting JSON into separate columns, creating a new calculated column, joining the newly ingested records with a static dimension table that is already in your database, etc). For these cases, using an update policy is a very common and powerful practice.

Each time records get ingested into the source table, the update policy's query (which we'll define in the update policy) will run on them (**only on newly ingested records** - other existing records in the source table aren't visible to the update policy when it runs), and the results of the query will be appended to the target table. This function's output schema and target table schema should exactly match.

Task 1: User defined Function (Stored Functions)

1. Create a stored functions, named ManiputatelogsRaw, that will contain the code below. Make sure the function works.
2. Replace all the code in the **cell** with the following code and click on ▷ **Run cell** button and review the output.

```
logsRaw
| where Component in (
'INGESTOR_EXECUTER',
'INGESTOR_GATEWAY',
'INTEGRATIONDATABASE',
'INTEGRATIONSERVICEFLOWS',
'INTEGRATIONSERVICETRACE')
```

Task 2: Create an update policy

In this task, we will use an update policy to filter the raw data in the logsRaw table (the source table) for ingestion logs, that will be ingested into the new table ingestionLogs that we'll create.

1. Build the target table,replace all the code in the **cell** with the following code and click on ▷ **Run cell** button and review the output.

```
.create table ingestionLogs (
Timestamp: datetime,
Source: string,
Node: string,
Level: string,
Component: string,
ClientRequestId: string,
Message: string,
Properties: dynamic)
```

2. Create a function for the update policy, replace all the code in the **cell** with the following code and click on ▷ **Run cell** button and review the output.

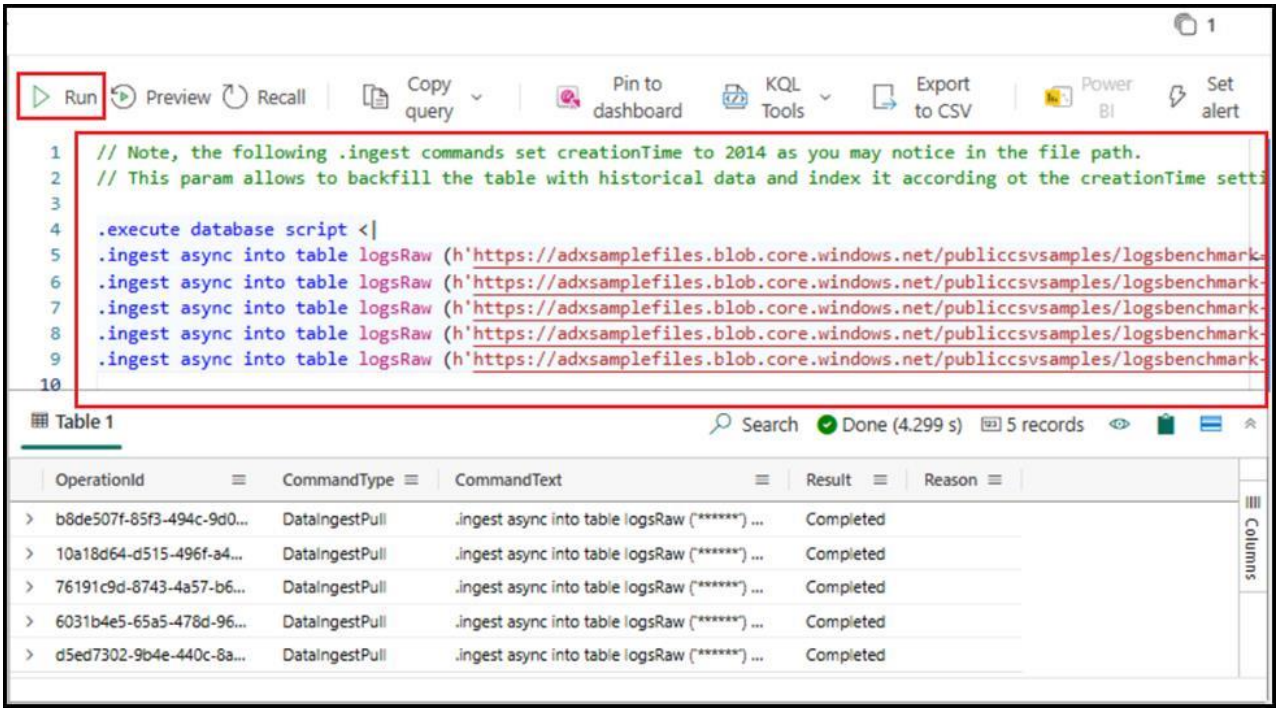
```
.create function ingestionComponents(){

logsRaw

| where Component has_any ('INGESTOR_EXECUTER', 'INGESTOR_GATEWAY',
'INTEGRATIONDATABASE','INTEGRATIONSERVICEFLOWS', 'INTEGRATIONSERVICETRACE', 'DOWNLOADER')

}
```

3. Create the update policy(Fill in the blanks), replace all the code in the **cell** with the following code and click on ▷ **Run cell** button and review the output.

```
.alter table ingestionLogs

   policy update  @'[{ "IsEnabled": true,"Source": "logsRaw", "Query": "ingestionComponents()",
"IsTransactional": true, "PropagateIngestionProperties": false} ]'
```

4. Update policy can transform and move the data from source table from the time it is created. It cannot look back at already existing data in source table. We will ingest new data into logsraw table and see new data flowing into ingestionLogs table
5. Replace all the code in the **cell** with the following code and click on ▷ **Run cell** button and review the output.
6. copy **operationIds** and paste them in a notepad , and then Save the notepad to use the information in the upcoming step.

```
// Note, the following .ingest commands set creationTime to 2014 as you may notice in the file path.

// This param allows to backfill the table with historical data and index it according ot the creationTime setting.


.execute database script <|

.ingest async into table logsRaw
(h'https://adxsamplefiles.blob.core.windows.net/publiccsvsamples/logsbenchmark-
onegb/2014/03/08/00/data.csv.gz') with (format='csv', creationTime='2024-03-08T00:00:00Z');

.ingest async into table logsRaw
(h'https://adxsamplefiles.blob.core.windows.net/publiccsvsamples/logsbenchmark-
onegb/2014/03/08/01/data.csv.gz') with (format='csv', creationTime='2024-03-08T01:00:00Z');

.ingest async into table logsRaw
(h'https://adxsamplefiles.blob.core.windows.net/publiccsvsamples/logsbenchmark-
onegb/2014/03/08/02/data.csv.gz') with (format='csv', creationTime='2024-03-08T02:00:00Z');

.ingest async into table logsRaw
(h'https://adxsamplefiles.blob.core.windows.net/publiccsvsamples/logsbenchmark-
onegb/2014/03/08/03/data.csv.gz') with (format='csv', creationTime='2024-03-08T03:00:00Z');

.ingest async into table logsRaw
(h'https://adxsamplefiles.blob.core.windows.net/publiccsvsamples/logsbenchmark-
onegb/2014/03/08/04/data.csv.gz') with (format='csv', creationTime='2024-03-08T04:00:00Z');
```



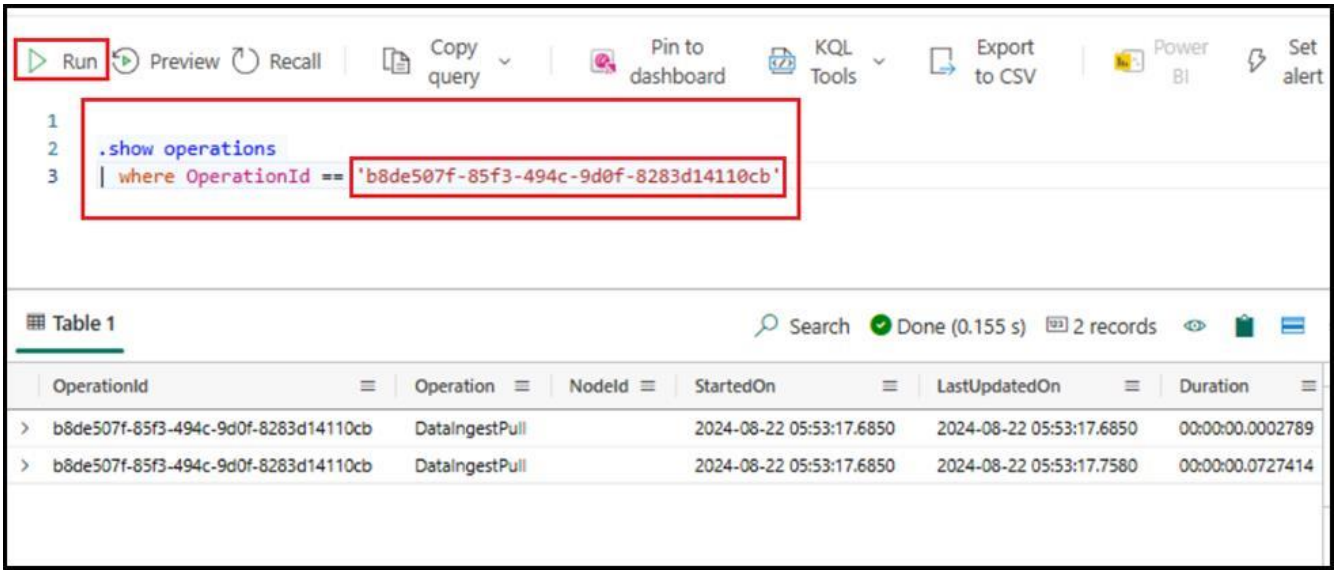
Note: The above command does not complete immediately. Because we used the async parameter, the output of the above query will be **operationIds**. The progress of the query can be checked by used the below command

7. Check progress of the commands, replace all the code in the **cell** with the following code and click on ▷ **Run cell** button and review the output.

.show operations

| where OperationId == ' OperationId '

Note: Replace the operationIds which you have saved in Step 6

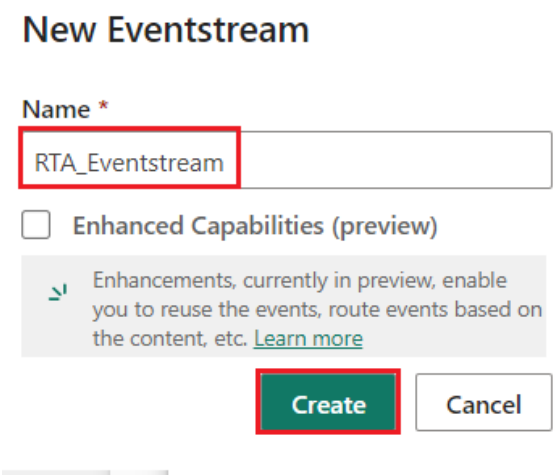


Note: If the count is not matching for ingestionLogs table, it means that one of the above .ingest commands have throttled or failed.

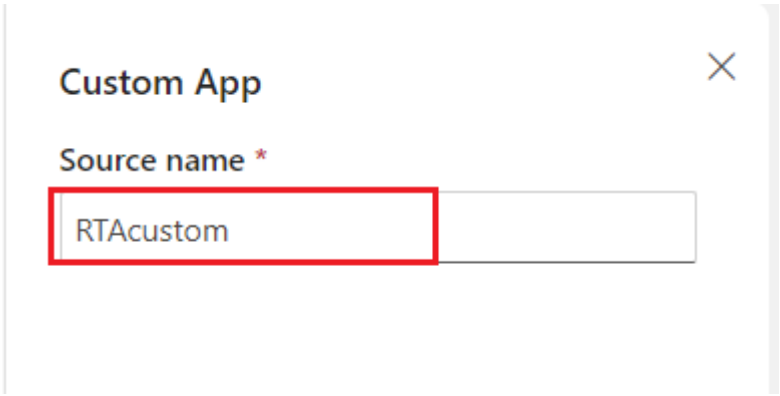
Exercise 5: Eventstreams

Task 1: Create Eventstream

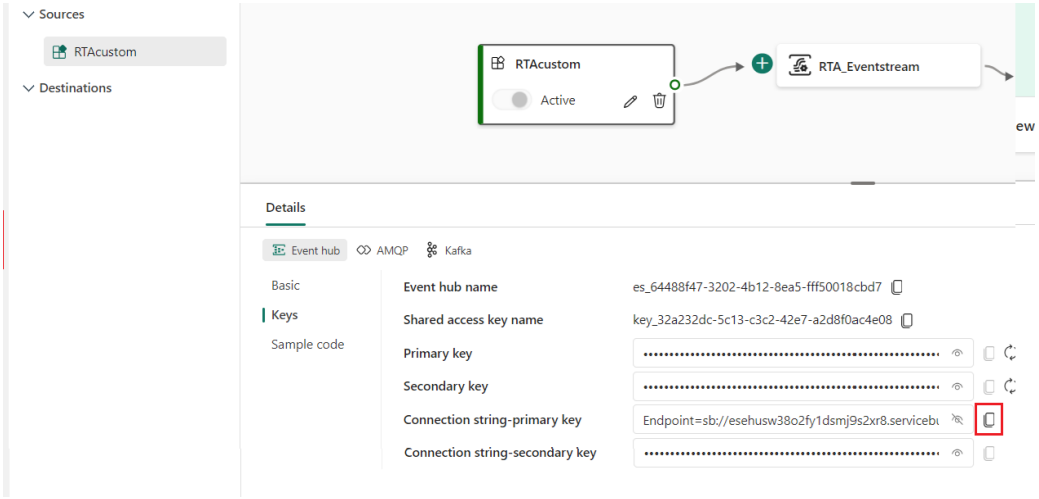
1. Select **your** *workspace* in the left navigation pane.
2. Select the **New** drop-down, and from there select **Eventstream.** Name the Eventstream **RTA_EventStream** and click on the **Create** button.



3. On the Eventstream, select **New source** and select **Custom App.**
4. On the **Custom App** configuration page, enter the source name as **RTAcustom** and click on **Add** button and publish the app.

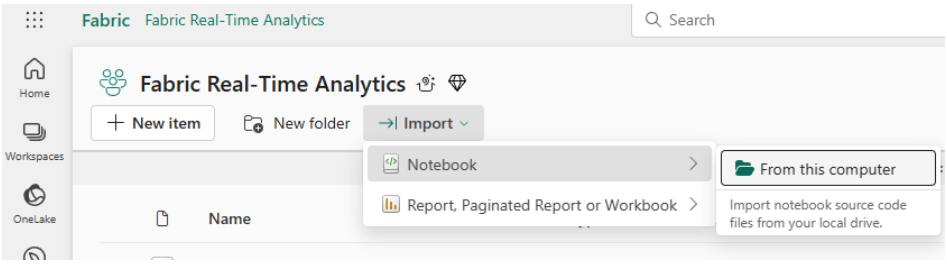


5. On the **Eventstream** pane, select the **keys** under the Details ,copy the **connection strings-primarykey** and paste them on a notepad, as you need them in the upcoming task

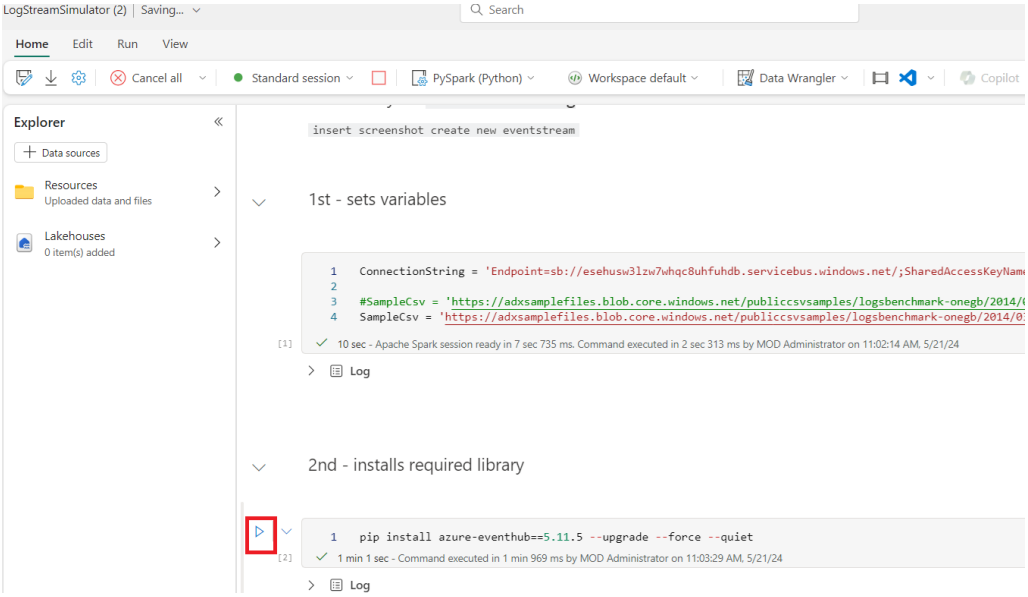


Task 2: Import LogSteamSimulator notebook

1. Download **LogSteamSimulator** notebooks.
2. Now, select **your** *workspace* in the left navigation pane
3. Select **Import notebook.**



4. Select **Upload** from the **Import status** pane that appears on the right side of the screen.
5. You will see a notification stating **Imported successfully.**
6. *Open*, the **LOgStreamSimulator** notebook.
7. To start the notebook, in the 1st cell paste the **connection string of your custom app source**(the value that you have saved in your notepad in the **Exercise 5>Task 1>Step 5)**, select the **Run** icon that appears on the left side of the cell.
8. To installs required library, select and run the 2nd , 3rd cells.



9. To sends the data, select and run the 4th ,5th cells.

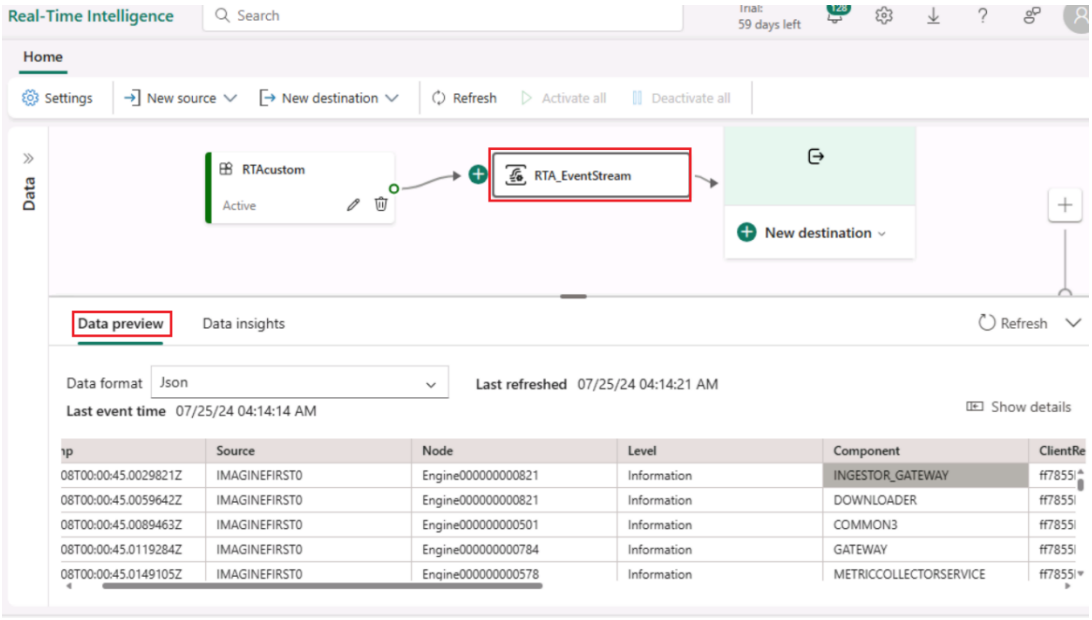
   Note: If you get AnalysisException: [PATH_NOT_FOUND] Path does not exist, replace code cell 1 with **backup URL location** https://github.com/microsoft/FabricRTA-in-a-Day/blob/main/assets/blobURIbackup.md#lab-3 for SampleCsv,

   Important Note: If either of the last code cells in step 4 fail with error file does not exist, then refresh the browser page and click stop session to get a new session (not reconnect to existing session), then click Run All again.

   Errors with pip's depency resolver are okay, proceeding code cells should run successfully.

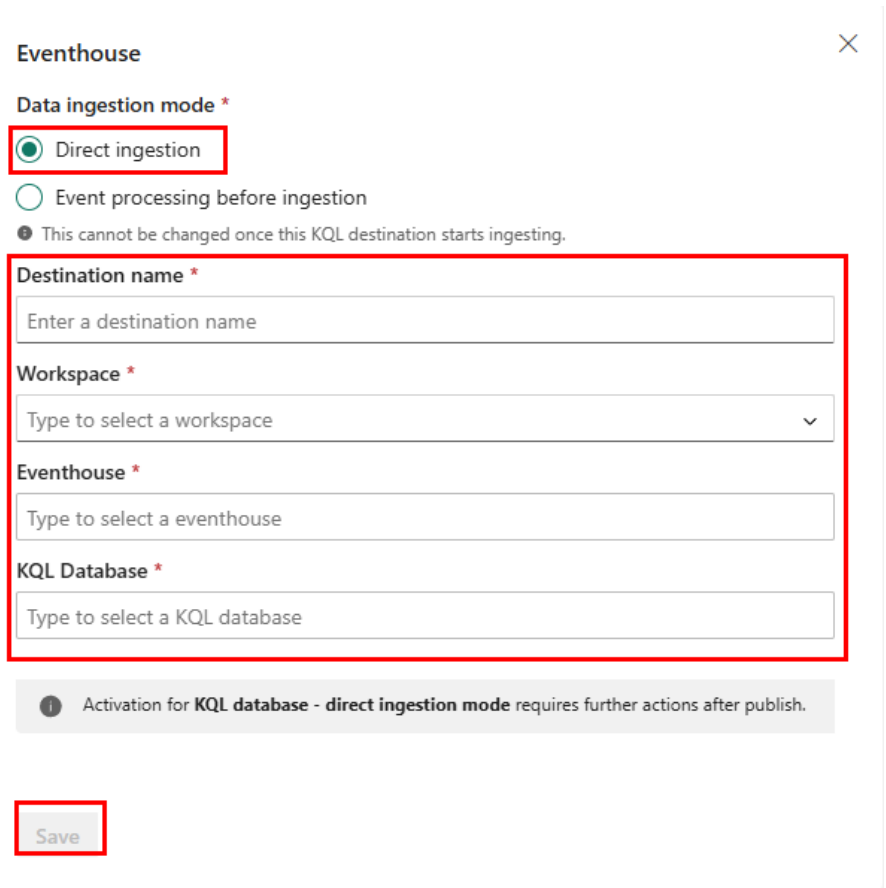
   ⚠ If you get ImportError: cannot import name 'Buffer' from 'typing_extensions', replace code cell 2 with: pip install azure-eventhub==5.11.5 --upgrade --force and click Run All again.

10. Select **RTA-Eventstream** in the left navigation pane.
11. Open eventstream artifact, preview the data stream

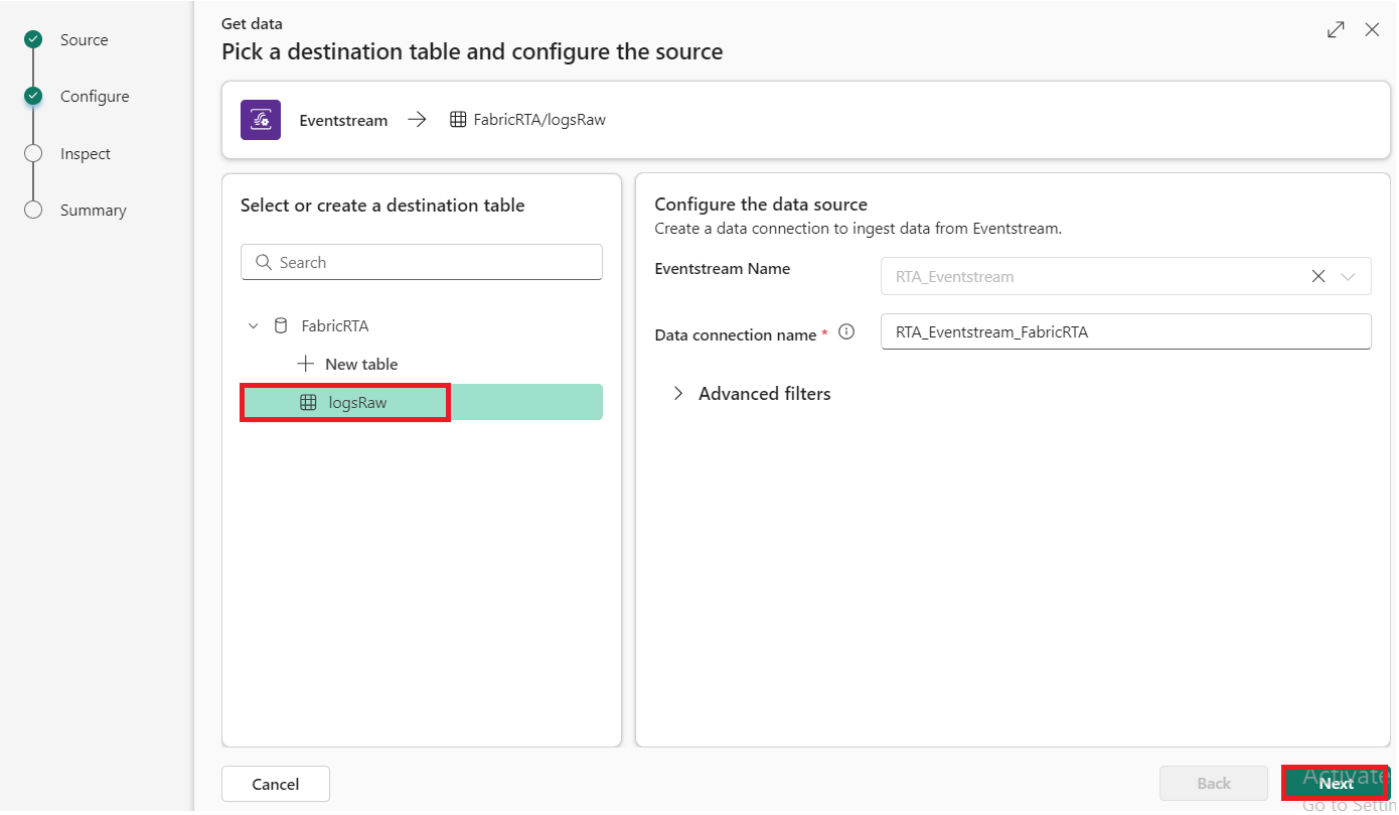


Task 3: Send data from the Eventstream to the KQL database

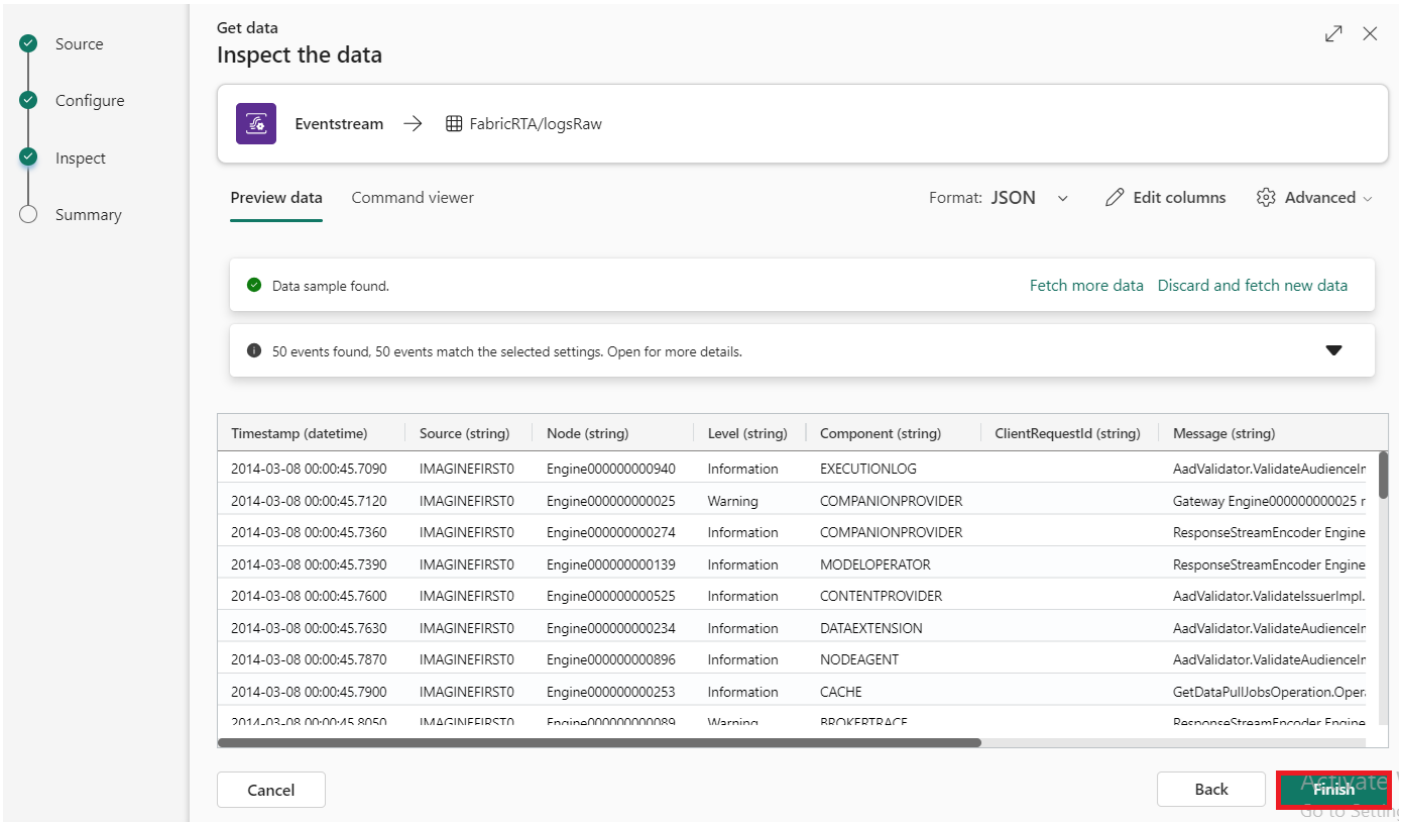
1. Our data should be arriving into our Eventstream, and we'll now configure the data to be ingested into the KQL database we created in the above task. On the Eventstream, click on **New destination**, then navigate and click on **Eventhouse**.
2. On the KQL settings, select **Direct ingestion**. While we have the opportunity to process event data at this stage, for our purposes, we will ingest the data directly into the KQL database. Set the destination name to **RTA-Destination**, then select your **workspace, Eventhouse,** and KQL database created in the above task, then click on **Save** button and **Publish.**



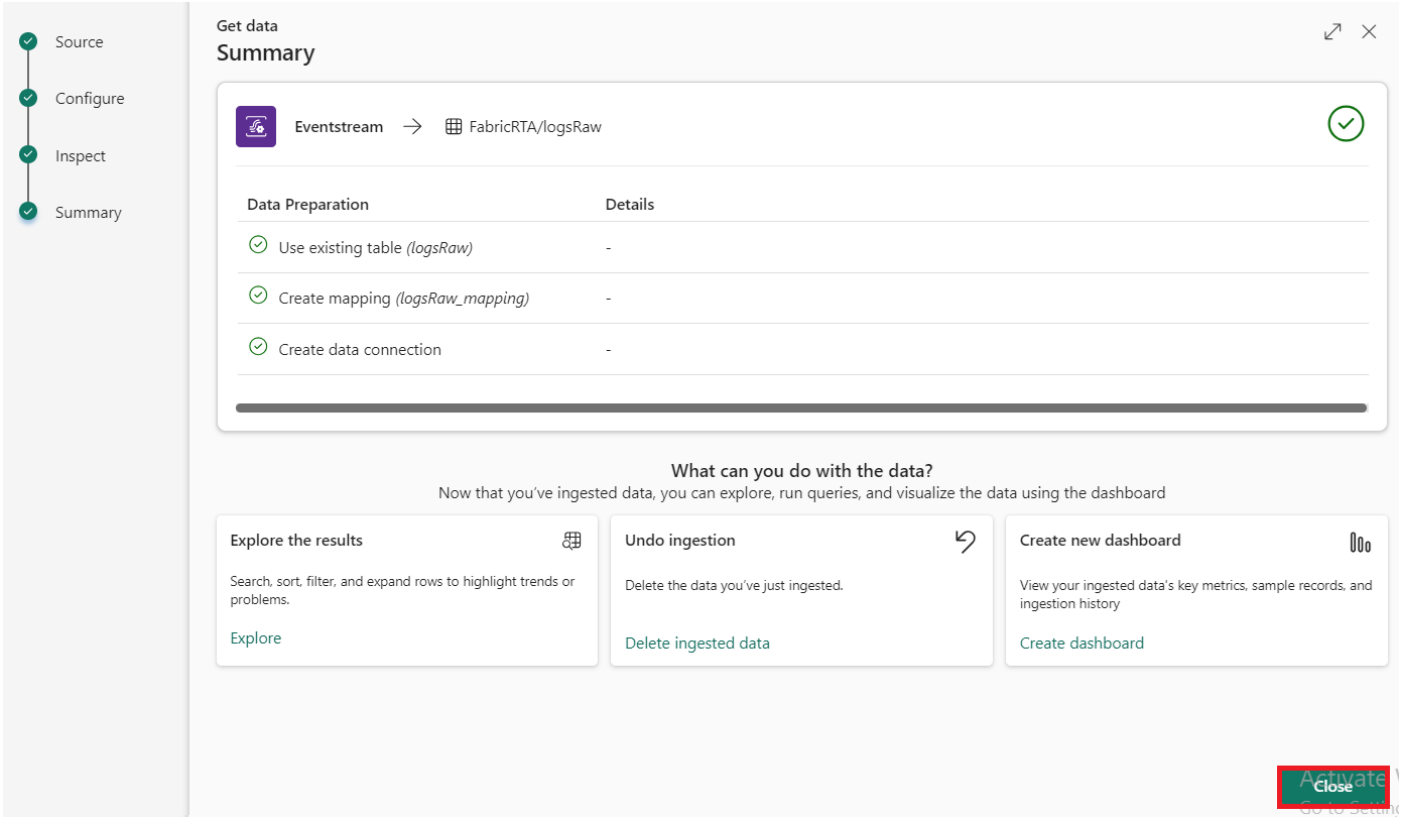
3. Once publish, select configure on destination. On the first settings page, select **logsRaw** table and click on the **Next** button.

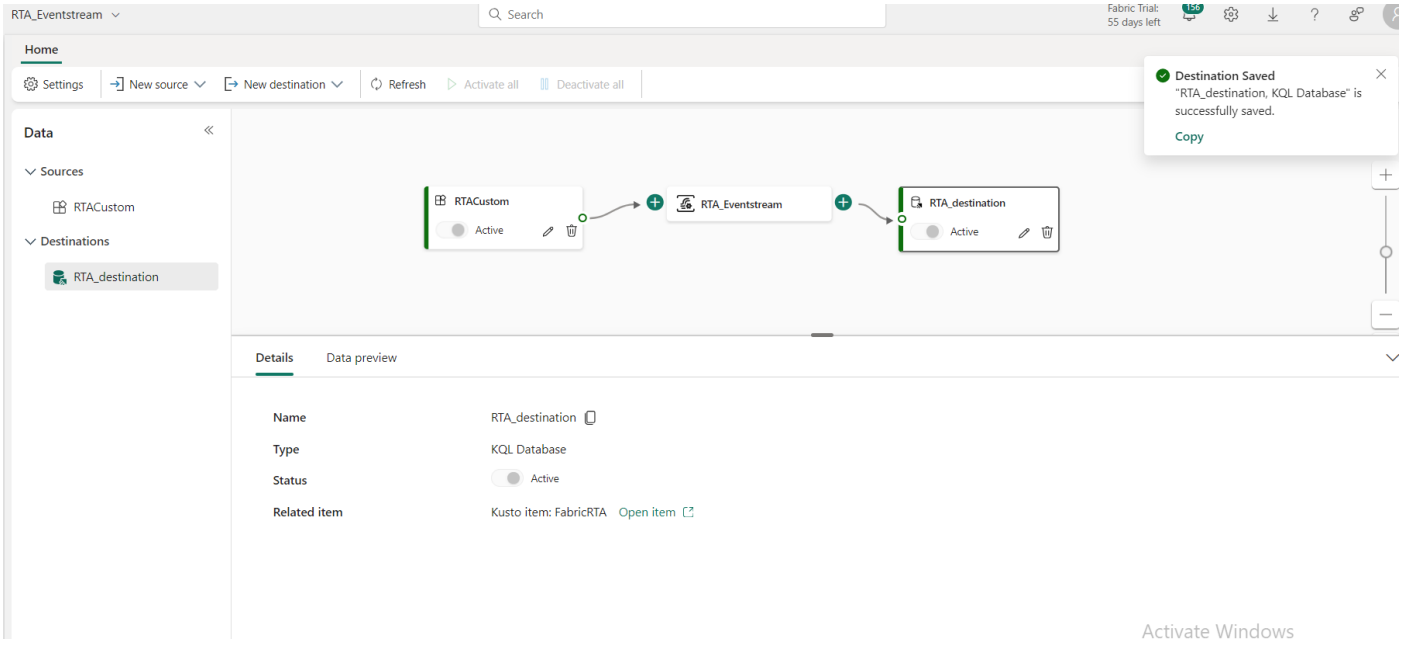


4. The next page allows us to inspect and configure the schema. Be sure to change the format from TXT to **JSON**, if necessary. The default columns of *symbol*, *price*, and *timestamp* should be formatted as shown in the below image; then click on the **Finish** button.



5. On the **Summary** page, if there are no errors, you'll see a **green checkmark** as shown in the below image, then click on the **Close** button to complete the configuration.

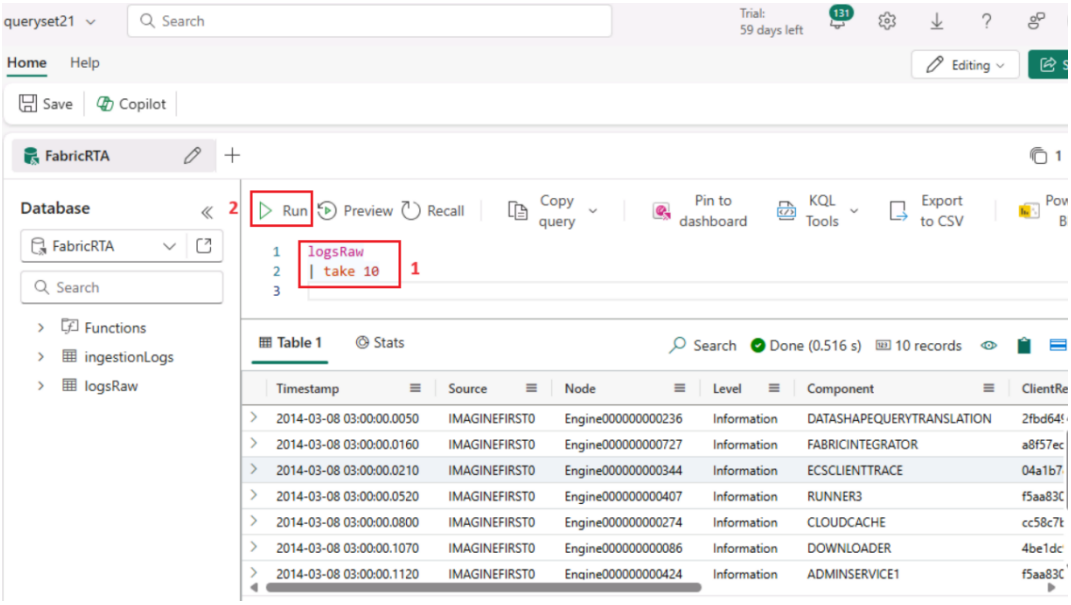




6. Now, select **your** *workspace* in the left navigation pane and open **KQL Queryset**

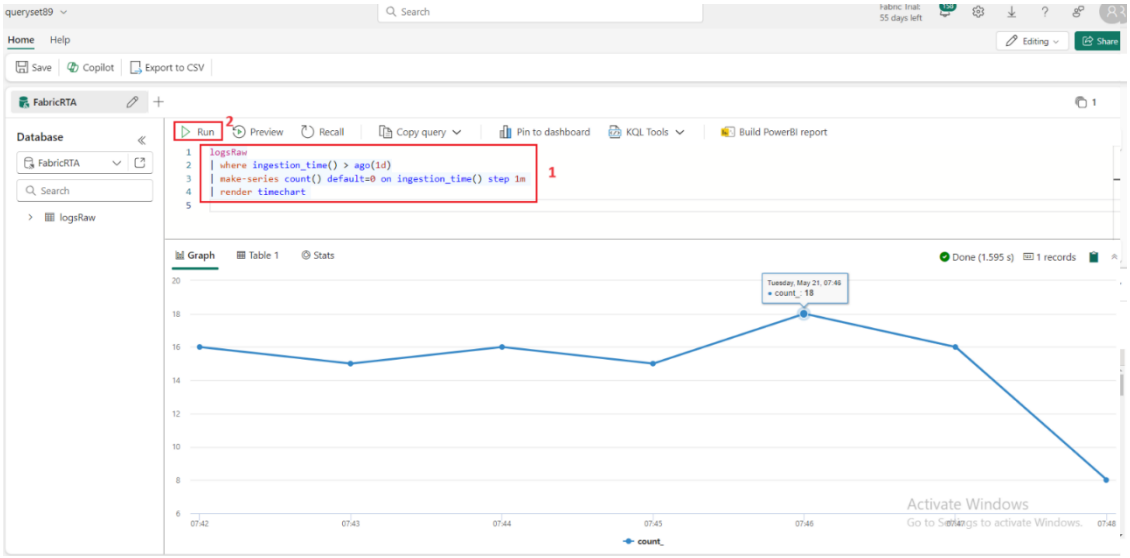
7. Replace all the code in the **cell** with the following code and click on ▷ **Run cell** button and review the output

```
logsRaw
| take 10
```



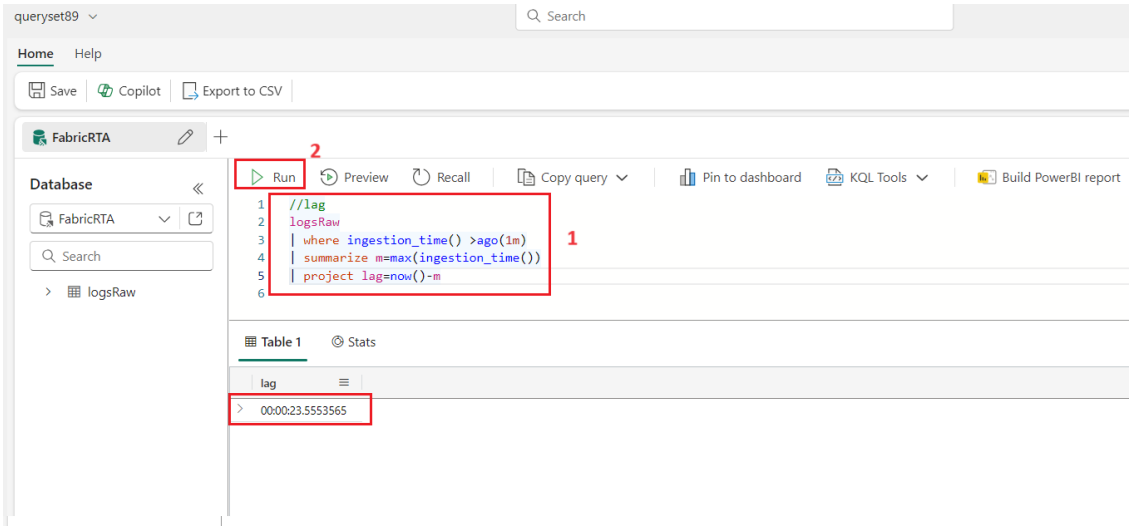
8. Replace all the code in the **cell** with the following code and click on ▷ **Run cell** button and review the output

```
//rate
logsRaw
| where ingestion_time() > ago(1d)
| make-series count() default=0 on ingestion_time() step 1m
| render timechart
```



9. Replace all the code in the **cell** with the following code and click on ▷ **Run cell** button and review the output

```
//lag
logsRaw
| where ingestion_time() >ago(1m)
| summarize m=max(ingestion_time())
| project lag=now()-m
```


queryset89
Search
Home
Help
Save
Copilot
Export to CSV
FabricRTA
Database
FabricRTA
Search
logsRaw
2
Run
Preview
Recall
Copy query
Pin to dashboard
KQL Tools
Build PowerBI report
1
//lag
logsRaw
| where ingestion_time() >ago(1m)
| summarize m=max(ingestion_time())
| project lag=now()-m
Table 1
Stats
lag
00:00:23.5553565