# Data Observability with Databand

Elena Lowery, WW Team, Data and AI

## Table of contents

## Contents

# Overview

In this lab you will learn how to use Databand to implement data observability for your data pipelines. You will complete the following tasks:

- Use Databand SDK in Python pipelines
- Use Databand SDK in PySpark pipelines
- Review lineage implementation in Databand

# Required software, access, and files

1. To complete exercises in this lab, you will need:

   1. A *Python IDE*
      - You can use any Python IDE with Python 3.8 or 3.9. We provide sample Python code/detailed instructions for
        - *Jupyter Notebooks* in Cloud Pak for Data as a Service (*CPDaaS*)
        - *JupyterLab* in Anaconda Community Edition
        - PyCharm Community Edition (can be installed with Anaconda Community Edition)

   The sample notebooks can also be used in other environments but may need small changes. We recommend that you use at least 1 notebook environment and at least 1 IDE, such as *PyCharm* or *Visual Studio*.

   2. A URL and a userid for the Databand demo environment (provided by the instructor).

3. Files from this Git repo.

# Part 1: Introduction to Databand SDK for data pipelines

Databand provides the capability to monitor data pipeline status and create alerts for pipeline and data quality issues. In this section you will learn how to use the *Databand Tracking SDK* to monitor Python pipelines. Databand also supports data pipelines written in Java, Scala, and dbt. You can find more information about support for these pipelines in [documentation](#).

1. Log in to the Databand environment provided by your instructor. If you're working on this lab on your own time, you can use this [environment](#).

2. Navigate to the **Pipelines** page and find the *prepare_sales_data* pipeline. You can filter by project – *Retail Analytics*.

   This simple data pipeline reads data from a csv file, filters out several columns, and writes 2 datasets for different product categories.



3. Click on the pipeline, then on one of the successful runs.

This view shows that the pipeline has three steps: *read_raw_data*, *filter_data*, and *write_data_by_product_line*.
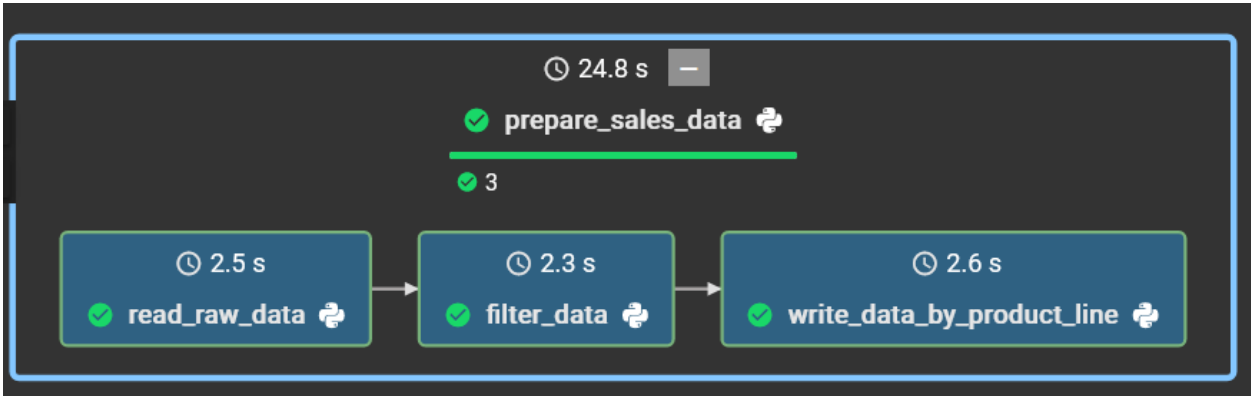


2. Click on the *read_raw_data* step in the pipeline, then switch to the **Metrics** tab.

The **Metrics** tab shows default and custom metrics that can be monitored for each step of the pipeline. The metrics on this page will be easier to understand after we review code for the pipeline.

Click **Show Schema** next to *Retail_Products_And_Customers.csv*. Here we see a preview of data that was read in the *read_raw_data* step of the pipeline.



Next, you will create a copy of this pipeline and explore the details of monitoring in Databand.

4.  Get an authorization token from Databand. The token is used in data pipelines when establishing a connection to Databand.

    In Databand click on **Settings**, then select the **Personal Access Tokens** tab. Click the **Create New Token** button. Provide a unique name (for example, add your initials), and save the token in a notepad.
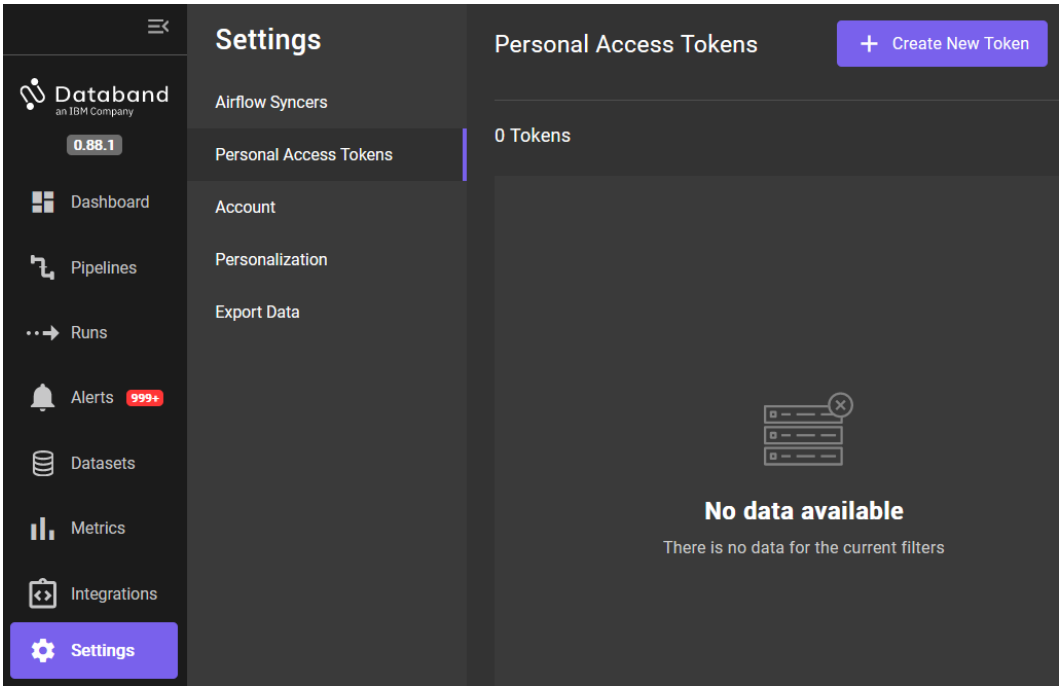
5. Load the sample data file and the Python script into your IDE. The samples are located in the *Workshop/Pipelines* folder (downloaded from the git repo or Box).

- Script: *SimpleRetailDataPipeline_with_Databand.py*
- Csv file*: Retail_Products_and_Customers.csv*
- *Notebook: SimpleRetailDataPipeline_with_Databand.ipynb*

In this lab we will review the *Python script*. If you're using the notebook example, see the instructions and comments in the notebook.

3. In the beginning of the script, we import pandas and Databand libraries. If your Python IDE does not automatically install libraries, then you can install them with pip:

- `pip install pandas`
- `pip install databand`

```
# Import pandas and databand libraries
import pandas as pd
from dbnd import dbnd_tracking, task, dataset_op_logger
```

4. Replace the *url* and *token* variables with the values for your Databand cluster.

```
# Import pandas and databand libraries
import pandas as pd
from dbnd import dbnd_tracking, task, dataset_op_logger


databand_url = 'insert_url'
databand_access_token = 'insert_token'
```

5. Replace the value of the *unique_suffix* variable to your initials.

Since we may have many workshop participants using the same cluster, adding a unique suffix to assets that are tracked in Databand will make it easier for your to find your pipelines and datasets.

```
# Provide a unique suffix that will be added to various assets tracked in Databand. We use this approach because
# in a workshop many users are running the same sample pipelines
unique_suffix = '_el'
```

6. Scroll down to the bottom of the script – to the *prepare_retail_data()* function.

Review the code:

- This function starts tracking the pipeline in Databand and invokes 3 functions that represent pipeline steps: *read_raw_data(), filter_data(), write_data_by_product_line()*. Notice that in Databand the names of the steps match the names of the Python functions in our pipeline.

```
def prepare_retail_data():
    with dbnd_tracking(
        conf={
            "core": {
                "databand_url": databand_url,
                "databand_access_token": databand_access_token,
            }
        },
        job_name="prepare_sales_data" + unique_suffix,
        run_name="weekly",
        project_name="Retail Analytics" + unique_suffix,
    ):
        # Call the step job - read data
        rawData = read_raw_data()

        # Filter data
        filteredData = filter_data(rawData)

        # Write data by product line
        write_data_by_product_line(filteredData)

        print("Finished running the pipeline")
```

- The main function, *prepare_retail_data,* starts tracking execution of the pipeline in Databand. Notice that the *job_name* corresponds to the *pipeline name* in Databand and *project_name* corresponds to the *project name*.



- Review the implementation of each function.

    o First, we add the *@task* decorator before the declaration of each function in Python. This lets Databand know that we are starting execution of a pipeline step. The *@task* decorator uses the name of the function directly below it as the name of the pipeline step in Databand.

    o Next, we log datasets that are used in the pipeline with the *logger.set()* call. This call will log *metadata* and a small set of sample data. If the customer has concerns about logging data, they can turn off logging the data sample.

    o In our example we read a csv file into a pandas dataframe, which is what's logged in Databand. Logging dataset metadata in Databand is optional, and it should only be used if data engineers need dataset metadata for troubleshooting issues. When you log datasets, you will see data in the **Metrics** tab of a pipeline step. If you switch back to Databand **Metrics** view

Data Observablity with Databand

6

of the *read_raw_data* step, you will notice that the schema and sample records match the few rows of the csv file.

```python
@task
def read_raw_data():
    retailData = pd.read_csv('Retail_Products_and_Customers.csv')

    # Log the data read
    with dataset_op_logger("local://WeeklySales/Retail_Products_and_Customers.csv", "read", with_schema=True,
                           with_preview=True) as logger:
        logger.set(data=retailData)

    return retailData
```

Next, we will run the code.

7. Save the changes you've made in the script and run it at least 5 times to generate some metrics data. You can run the script in debug mode if you would like to review the SDK in more detail.

- As you are running the script, switch to the Databand environment, and monitor pipeline execution. You can find the pipeline in the **Pipelines** view.



- Drill down to see the details of the run.

6. Click on the **Metrics** tab, then select each step of the pipeline.

   Explore the **Metrics**, notice that the schema corresponds to the dataset read or written in each step.

   For example, the *Retail_Products_And_Customers.csv* dataset (in Python the *retailData* pandas dataframe) has 88,475 rows and 24 columns.
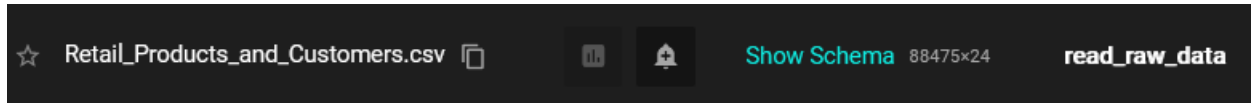
   

   If you click on the *write_data_by_product_line* step, you will notice that one of the output datasets, *Golf Equipment* (in Python the *GoflEquipment* pandas dataframe), has 7764 rows and 21 columns.
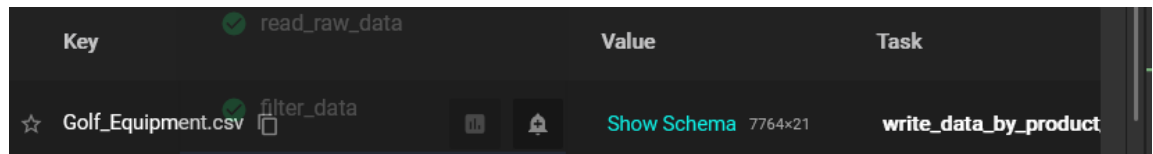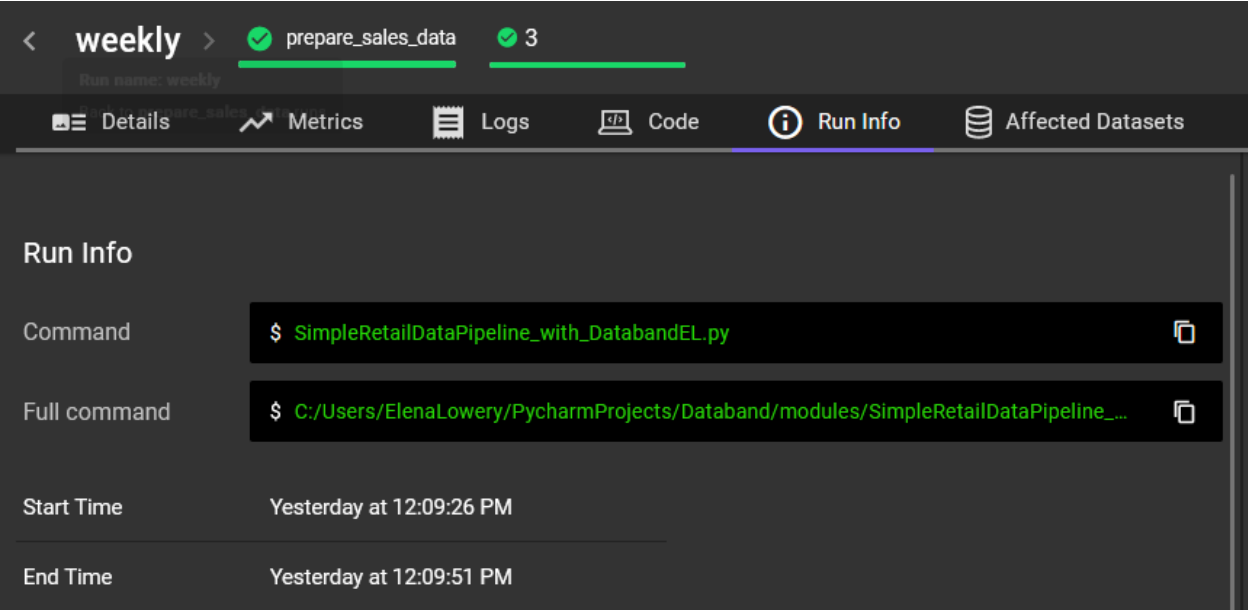
   

7. Click on the **Code** tab, then select each step of the pipeline.

   Here we can review (but not edit) code that corresponds to the pipeline step. Databand is not used for editing code, code changes should be done in the IDE that's used to develop pipelines.
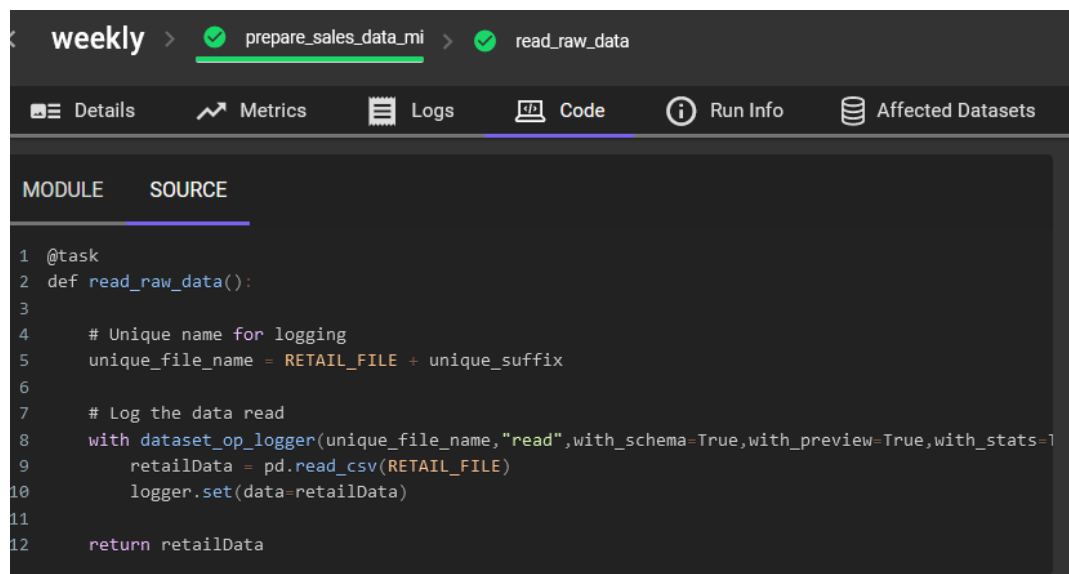
   

8. Click on the **Run info** tab and notice that it captures the name of the Python script that implements the pipeline.

9. Click on the **Affected Datasets** tab and the "top" pipeline view (you should see a bold blue box around the entire pipeline, and not the individual steps). Here we have a single view of all datasets used by the pipeline. Since we don't have any issues yet, we don't need to review additional details at this time.
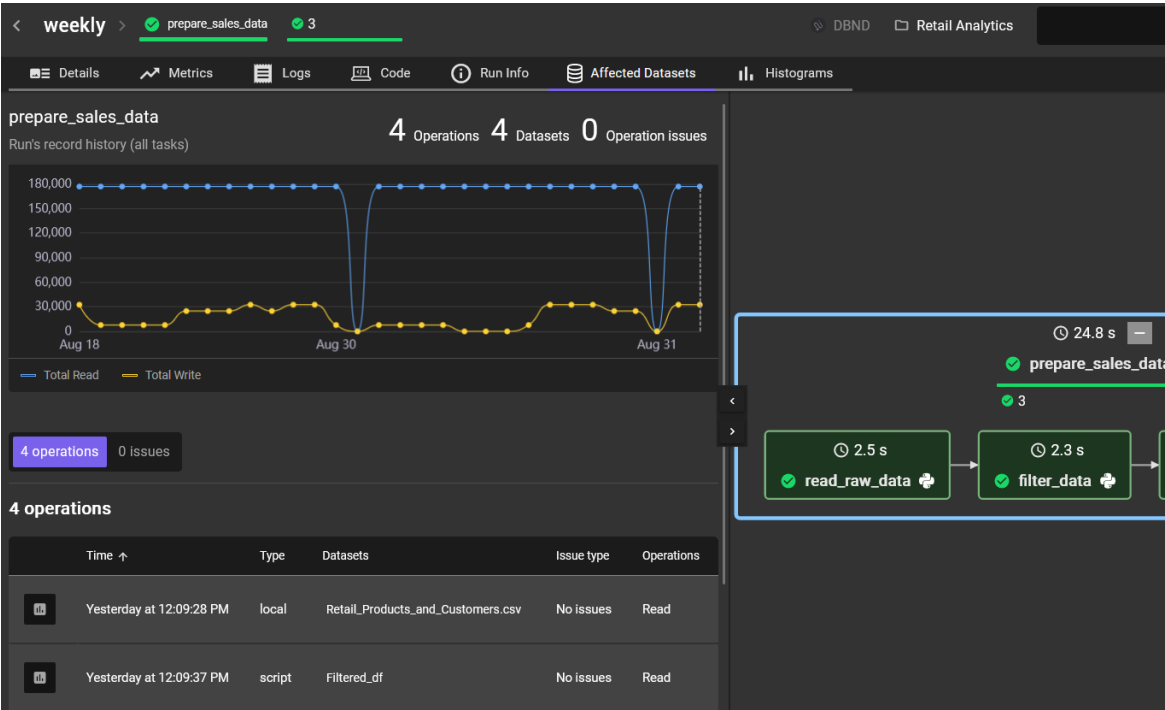


Next, we will introduce a few errors in the pipeline to understand how Databand can help us monitor and troubleshoot the errors.

We will start with introducing an error in the last step of the pipeline.

*Note: In this lab we are adding errors that are typically resolved at **build time** because it's the easiest way to create an error in code. In a production environment **runtime errors** will be handled the same way by Databand.*

10. In the *write_data_by_product_Line()* function find the line of code that writes the equipment csv and change it to the name of the pandas dataframe that doesn't exist, for example, *golfEquipment1*.

    *Note: While PyCharm shows the error, the code will still run because Python is not a compiled language.*

```
# Select any product line
golfEquipment = filteredData.loc[filteredData['Product line'] == 'Golf Equipment']

# Log the filtered data read
with dataset_op_logger(unique_file_name_2, "write", with_schema=True,
                       with_preview=True) as logger:
    # Write the csv file
    golfEquipment1.to_csv("Golf_Equipment.csv", index=False)

    logger.set(data=golfEquipment)
```

11. Save the change and run the script. Switch to Databand and find your pipeline in the **Pipelines** tab.

    Databand shows that the 3rd step of the pipeline failed and displays the error that caused the failure.



12. Switch to the **Alerts** view and click **Add Alert**.

    Create an alert for **Run State** *Failed*, and make it a *Critical* severity alert.

Run the pipeline again and refresh the **Alert** view. You should see a critical alert for your failed pipeline run.



13. Drill down to the alert and review the details.

As on the **Pipelines** page, we can see the error that caused the failure. We can also review the datasets that could be potentially affected by the error in the pipeline step.

During the first few failed runs of a new pipeline that's observed by Databand, you may see the message "*Unable to determine alert impact*." This happens because the impact analysis job did not run yet. If you see this message, return to the alert after 10-15 minutes. You can continue with the rest of the instructions in the lab.

Databand is able to determine (infer) the list of affected datasets because
- We used the *logger.set()* function in our data pipeline
- We had several successful runs of the pipeline, and these datasets were read or written in the successful runs.



Since the error in a Python function can happen before or after the dataset is written, we need to investigate if the expected number of rows was written to the dataset.

14. Click on *Camping_Equipment.csv*, then the **History** tab.

Based on this output, we have a consistent number of rows/columns that are being written to this dataset, including during the pipeline run that has failed (the last run shown on top of the table). That means that even though the pipeline run failed, data was written successfully.

This output aligns with the error we introduced in the code – it's after we write the *CampingEquipment.csv*.

| | Status | Type | Issue type | Schema | Records | Origin |
|---|---|---|---|---|---|---|
| ⋮ | ✅ | Write | No issues | 21 columns | 24,866 records | ⌐ prepare_sales_data EL  › <br> ⋯→ weekly  › <br> 🗋 write_data_by_product_line  › |
| ⋮ | ✅ | Write | No issues | 21 columns | 24,866 records | ⌐ prepare_sales_data EL  › <br> ⋯→ weekly  › <br> 🗋 write_data_by_product_line  › |

*Overview | History | Operations*
*Seen during: Last 7 days | Operation type: All | Pipeline: All | Issue type: All*
*22 Operations*

Next, we will introduce the error earlier in the code.

15. In the *write_data_by_product_line()* function find the line of code that references the pandas dataframe that was passed into the function, and change it to a different name, for example, *filteredData1*.

    Since *filteredData1* does not exist in this function, we will get an error.

    Run the script and switch back to Databand.

```
@task
)def write_data_by_product_line(filteredData):
     # Select any product line - we will write it to a separate file
 ❗  campingEquipment = filteredData1.loc[filteredData['Product line'] == 'Camping Equipment']
```

16. From the **Alerts** page, navigate to your failed pipeline run.

    Notice that in addition to the **Affected datasets** tab, we now see **Missing operations** tab. A "missing operation" means that the code that writes datasets did not run. Databand knows that a successful execution should result in writing the *Camping Equipment* and *Golf Equipment* datasets because we had several successful runs of the pipeline.

    In general, Databand *infers* affected datasets and pipelines by observing successful pipeline runs, which means that we should always ensure that a pipeline runs successfully several times after we configure it to be observed by Databand.

Next, we will review what happens if we have errors earlier in the pipeline.

17. Add an error in the 1st step of the pipeline. For example, change the name of the csv file to the name of the file that does not exist (global variable). Save the script and run it.

```
# Data used in this pipeline
RETAIL_FILE = "https://raw.githubusercontent.com/elenalowery/data-samples/main/Retail_Products_and_Customers1.csv"
```

Switch to Databand and find your failed pipeline run (from the **Pipelines** view).

Since the pipeline failed on the first step, the other steps are not shown.



We can see that more datasets are shown in the **Missing Operations** tab of the pipeline alert.

Next, we will configure 2 other types of alerts that are not pipeline failures, but nevertheless could signal a problem with the pipeline or data.

One of the most frequent alerts that a data engineering team is interested in is the *run duration* alert because it can indicate a problem with the pipeline.

18. In Databand switch to the **Alerts** panel and click **Add Alert**.



19. Select your pipeline.



20. Select **Run Duration (seconds)**, then **>** ,and enter the value that's smaller than the shortest duration of your pipeline runs.

In our example the longest successful run is *17.3* seconds, that's why we are specifying *15* seconds because we want to "force" the alert. Select any alert severity, then add a name and a description.

Notice that we have many options for creating alerts on this page. At this time we will focus on the overall pipeline duration. Click **Save**. Optionally, assign receivers for the alert.



21. Run your pipeline and check the **Alerts** page.

22. Drill down to the alert to view the details.
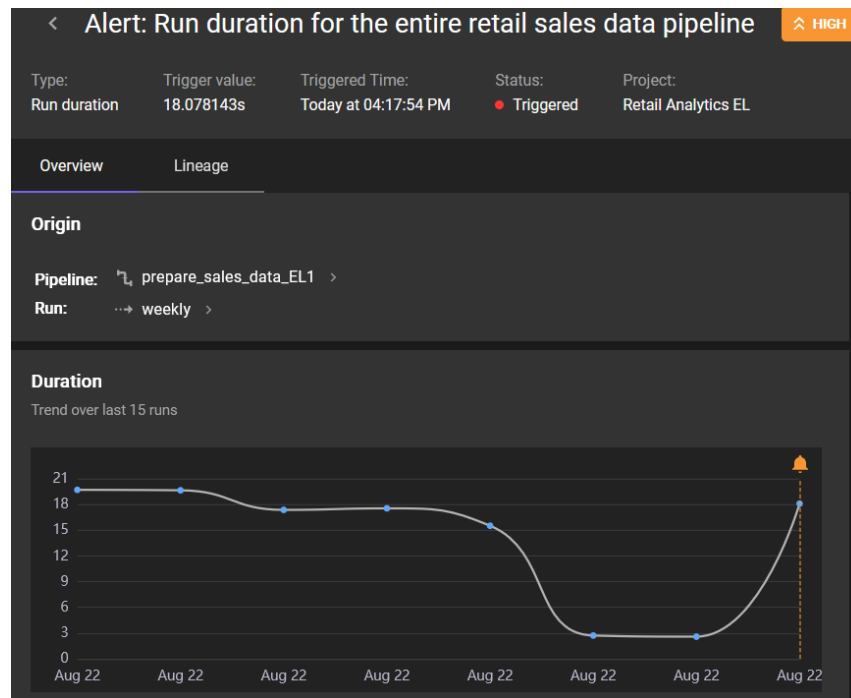


Now that you've tested the alert, either delete it or change the run duration to a higher value so that it does not generate too many unnecessary alerts in the workshop environment.

Next, we will create an alert for the *number of rows written* to the output dataset. We will use a different navigation approach to get to the alert definition page.

23. Navigate to your pipeline from the **Pipelines** panel, drill down to one of the successful runs, and make sure that the entire pipeline is selected (blue bold box around the entire pipeline).

Click on the **Metrics** tab.
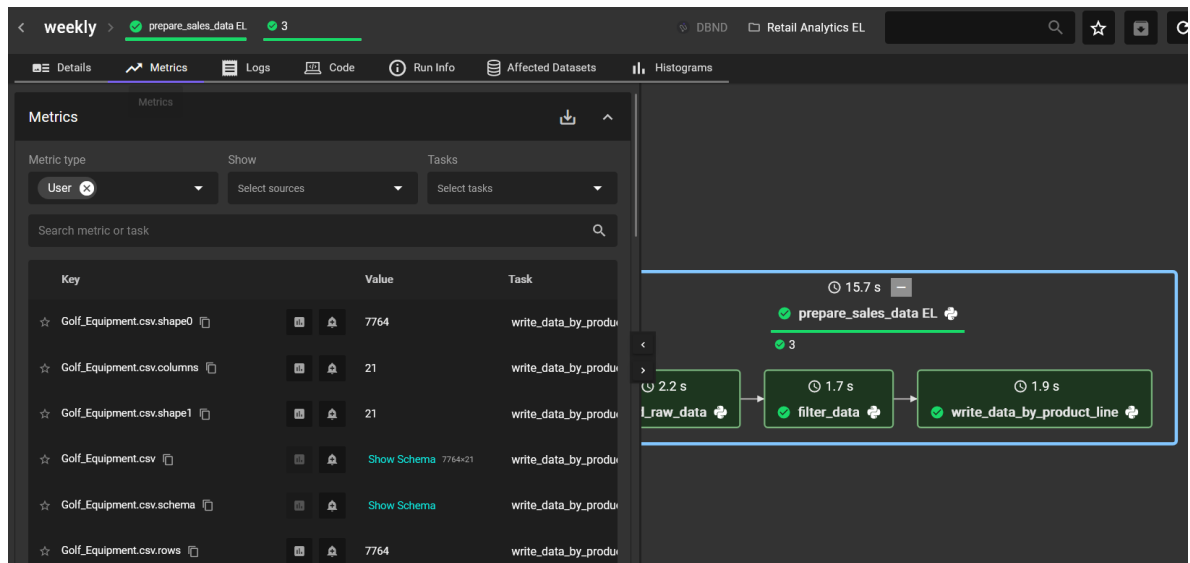
Click on the **Show on chart** icon next to *Camping_Equipment.csv.rows* and review the number of rows that has been written up to this point (scroll down to see the chart). Since we did not have runtime errors, the number of rows has consistently been 24,688. As a reminder, we created errors that resulted in a "missing operation".
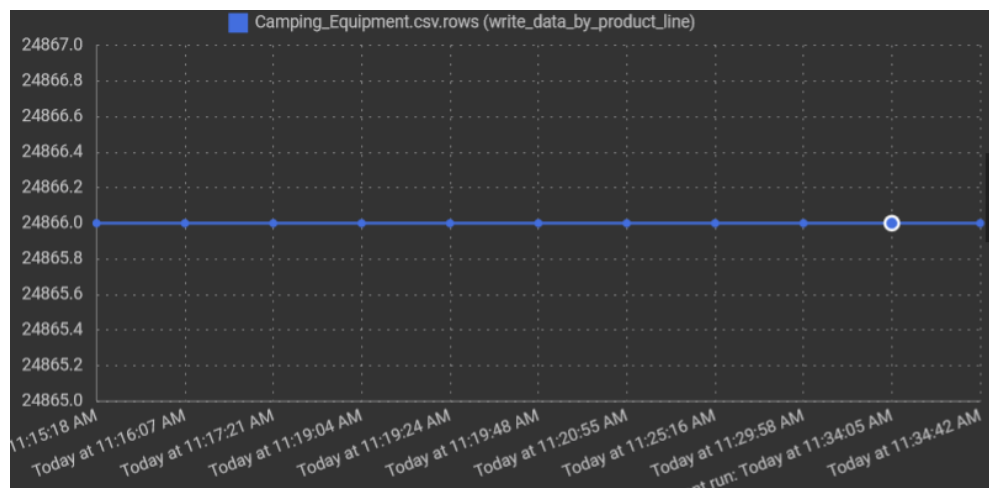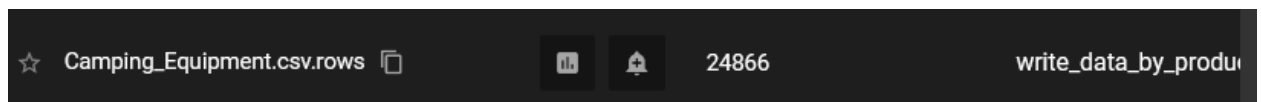




24. Click the **Create Alert** button next to *Camping_Equipment.csv.rows* (the bell icon). Notice that when the **Create alert** panel is displayed, the number of rows is pre-selected.

    You can choose whether to create an alert based on a hardcoded value or an anomaly. We decided to create an alert for the number of rows less than 10,000.

If you don't want to change the pipeline code to test this alert, then simply specify the number less than 24,688 to create the alert. However, we recommend that you change the code to test the alert.

Select the *Camping_Equipment.csv_write_rows* as the metric.



In the *write_data_by_product_line()* change the line of code that selects records from the pandas dataframe to a different valid value, for example, *Outdoor Protection*. The number of records for this filter will be different. Save the script and run it.

```python
@task
def write_data_by_product_line(filteredData):
    # Select any product line - we will write it to a separate file
    campingEquipment = filteredData.loc[filteredData['Product line'] == 'Outdoor Protection']
```

Next, change the value to an invalid value, for example, *Camping Equipment1*. The number or records should be 0. Save the script and run it.

```python
@task
def write_data_by_product_line(filteredData):
    # Select any product line - we will write it to a separate file
    campingEquipment = filteredData.loc[filteredData['Product line'] == 'Camping Equipment1']
```

25. Switch to Databand and find the generated alerts.

26. Investigate the issue through **Datasets** view.



Find the dataset, drill down and click on the **History** tab. Notice that the number of records written in the last 2 pipeline runs is different.



Next, we will add a *custom metric* to monitor data quality and set up an alert for it.

In Databand a *custom metric* is *business logic* that's implemented in a data pipeline and registered in Databand. Custom metrics can be used to monitor/alert for data quality or simply patterns in data that need to be investigated.

Let's start by reviewing the built-in metrics for data quality.

27. In Databand navigate to the **Alerts** page. Click **Add Alert -> Data** and select **Data Quality Check**.
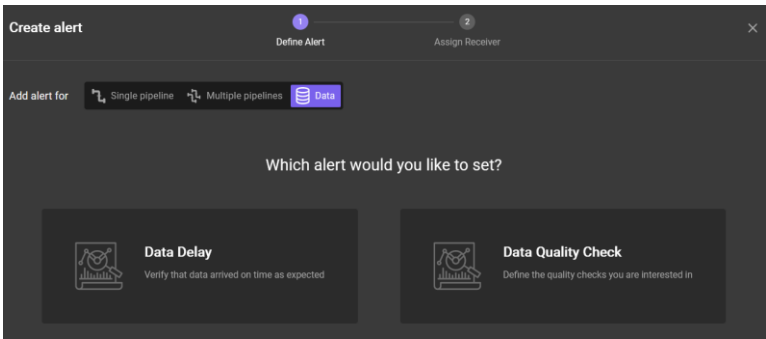


28. Select the *Camping Equipment* dataset and explore quality alerts that you can set up with the options available in the UI.

    Notice that we can select columns in the dropdown and set up alerts for *null* records, *min* and *max* values, *distinct counts*, etc.
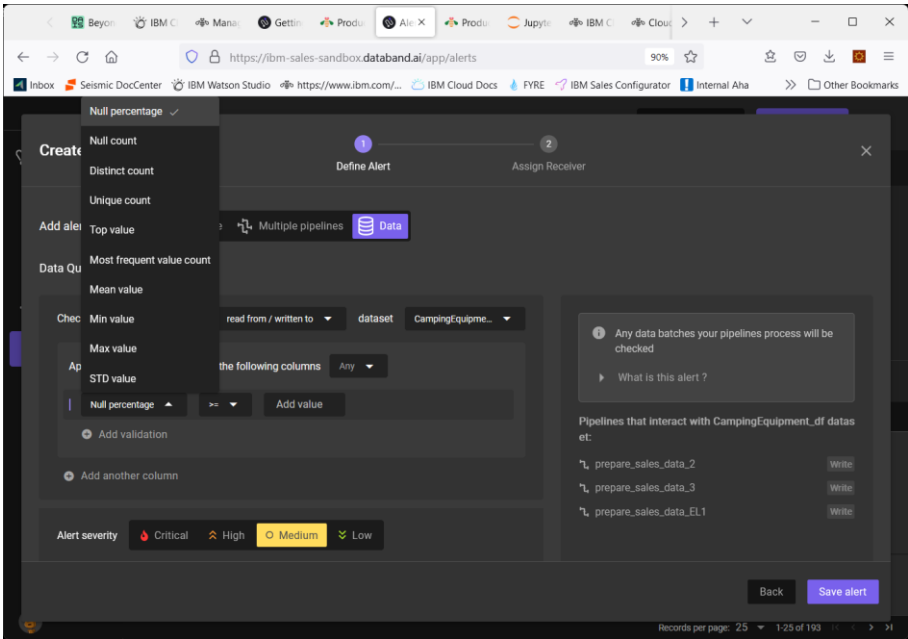


While the built-in options cover the most important data quality checks, we may have use cases that require custom logic.

For example, in our retail use case we want to issue an alert if no sales have been reported from a particular state. In order to do this, we will need to create a custom metric in our data pipeline.

We added this function to our data pipeline:

```python
def check_camping_equipment(rawData):

    metric_name = 'Sales from Alaska' + unique_suffix

    numberOfCampingEquipment_records = rawData['State'].tolist().count('Alaska')
    print(numberOfCampingEquipment_records)

    log_metric(metric_name, numberOfCampingEquipment_records)
```

In this example the *log_metric()* function sends the number of records for the specified state to Databand.

*Note: we hardcoded the state value for simplicity. In a production implementation, it's possible to make the values configurable so that states are changed without changing the code.*

29. Add the metric logging code to your *SimpleRetailDataPipeline_with_Databand* script or open *SimpleRetailDataPipeline_with_CustomMetric.py* script (from the *Workshop/Pipelines* folder)

If you're updating the original script, in addition to adding the function, add the function call to the *prepare_retail_data()* function (at the end).

```python
def check_camping_equipment(rawData):

    metric_name = 'Sales from Alaska' + unique_suffix

    numberOfCampingEquipment_records = rawData['State'].tolist().count('Alaska')
    print(numberOfCampingEquipment_records)

    log_metric(metric_name, numberOfCampingEquipment_records)
```

```
def prepare_retail_data():
    with dbnd_tracking(
        conf={
            "core": {
                "databand_url": databand_url,
                "databand_access_token": databand_access_token,
            }
        },
        job_name="prepare_sales_data" + unique_suffix,
        run_name="weekly",
        project_name="Retail Analytics" + unique_suffix,
    ):
        # Call the step job - read data
        rawData = read_raw_data()

        # Filter data
        filteredData = filter_data(rawData)

        # Write data by product line
        write_data_by_product_line(filteredData)

        check_camping_equipment(rawData)

        print("Finished running the pipeline")
```

30. Run the updated script several times with different state values (keep the metric names the same).

For example, filter for *Alaska* (no records), *Arizona*, and *Oregon*. We are providing different state names just to generate a different number of records in Databand.

```
def checkCampingEquipmentData(rawData):

    numberOfCampingEquipment_records = rawData['State'].tolist().count('Oregon')
    print(numberOfCampingEquipment_records)

    log_metric('Sales from Alaska', numberOfCampingEquipment_records)
```

31. In Databand navigate to the **Metrics** page and find *Sales from Alaska* metric.

Notice the number of records recorded on each run.

| | | Sales from Alaska | User | prepare_sales_data_3 > | | 259 |
| --- | --- | --- | --- | --- | --- | --- |
| | | | | prepare_sales_data_3 > | | |
| | | GolfEquipment_df.Retailer country.type | Histograms | prepare_sales_data_3 > | No graph | str |
| | | | | writeDatabyProductLine > | | |
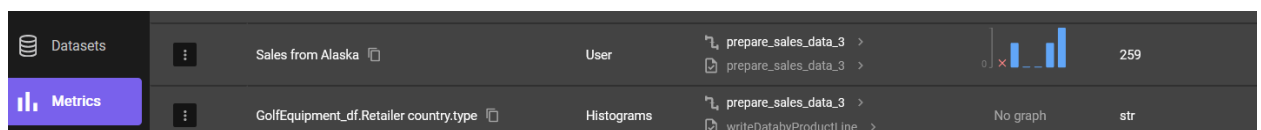
Drill down into the metric, then create an alert for number of records *== 0*.
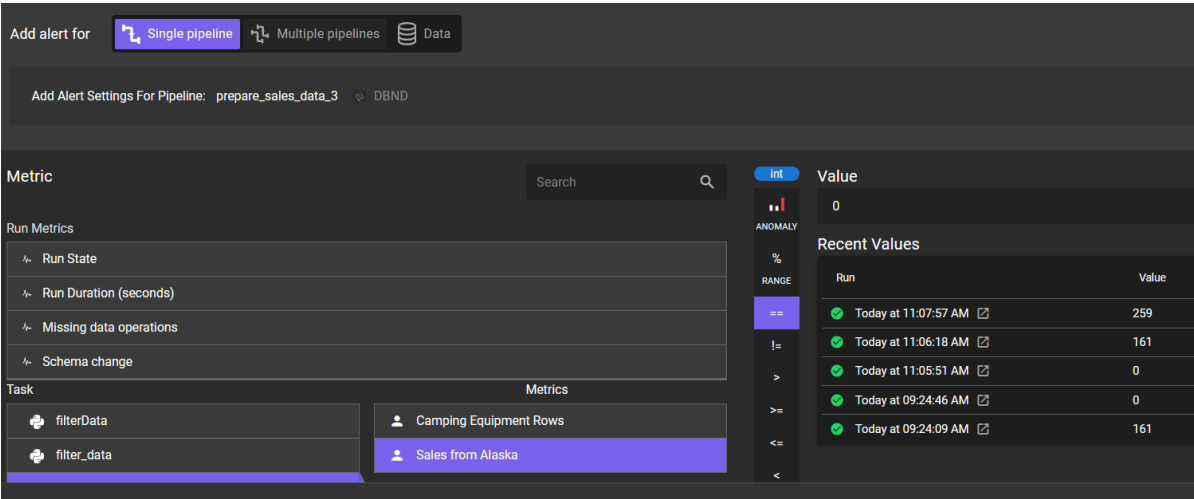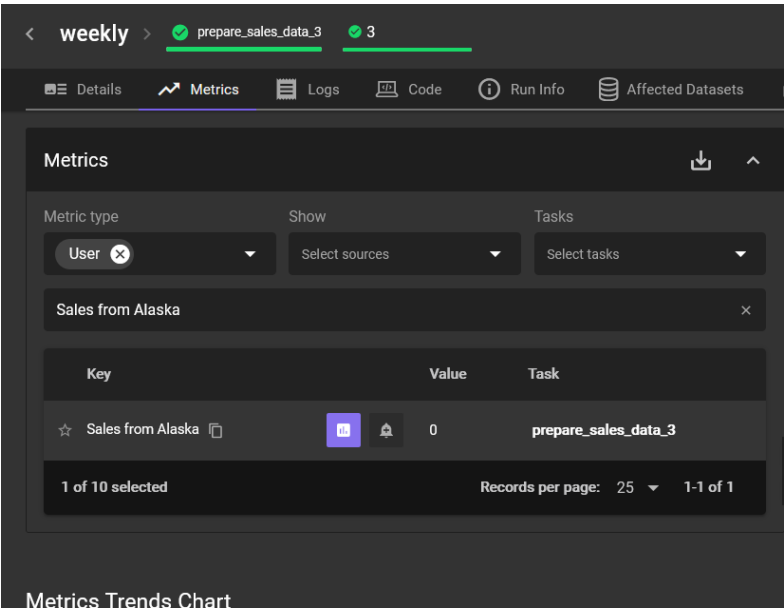
32. Change the pipeline to count records from *Alaska* again and run it.

    Switch to Databand and find the generated alert.
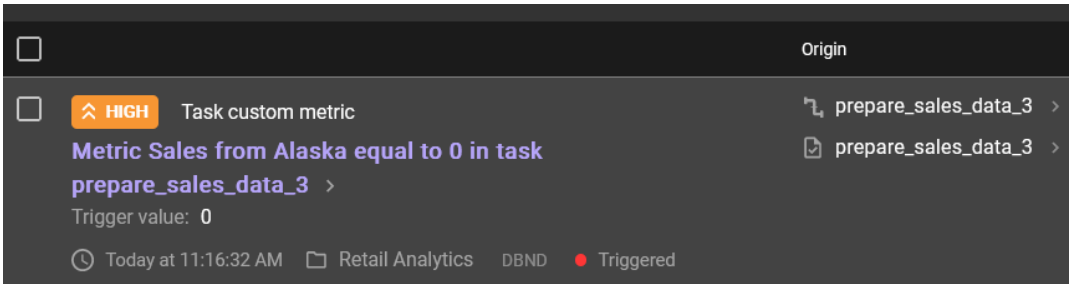


    This concludes introduction to Databand Python SDK.

**Summary:**

In this section you learned how to use Databand SDK to:
- Track Python pipelines in Databand
- Create alerts for data quality
- Create custom metrics.

These tasks are the most frequently used tasks in a production implementation of Databand.

# Part 2: Review a PySpark example

In this section you will use the Databand SDK to track execution of a Spark pipeline, specifically a PySpark notebook in *IBM Cloud Pak for Data*. Databand also supports monitoring PySpark and Scala pipelines that run in other distributions of Spark, such as *Databricks*, *Amazon EMR*, *Google Cloud DataProc,* and others.

In this section we will review programmatic integration with Spark pipelines that's similar to Python integration.
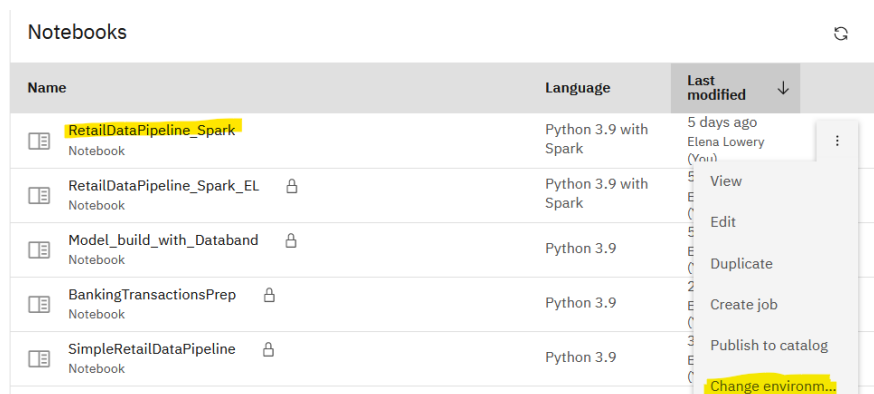
***Important note:*** *It's also possible to configure a Spark Listener that will automatically log datasets and track pipeline execution, enabling "no code" integration of Spark pipelines with Databand. See* [documentation](#) *for configuration information.*

The PySpark example that we provided for this section is a PySpark notebook that performs ETL operations on the same retail dataset as Python example we reviewed in **Part 1**. We recommend that you use *Cloud Pak for Data as a Service* (*CPDaaS*) as the runtime environment.

If you use another Spark environment, you can use the same sample notebook, but you may need to make changes that are specific to your Spark environment (reading/writing data).

1. Log in to *CPDaaS* and create a new project.

2. Import the *Retail_Products_and_Customers.csv* from the */Pipelines* folder of the file downloaded from Box.

3. Create a notebook from file: *RetailDataPipeline_Spark* (in the */Pipelines* folder). Make sure to set the Environment to *Default Spark 3.2 & Python 3.9 runtime.*

    If you forget to select the Spark environment, you can do it later by stopping the notebook and changing the default environment from the Project view (select the vertical ellipses menu next to the notebook). To unlock the notebook, click on the lock icon.

4. Follow the update instructions in the notebook:

    1. Generate the project token
    2. Replace code to read data from your object storage
    3. Provide Databand URL and token
    4. Rename the pipeline

5. Run the notebook and check Databand for pipeline status.
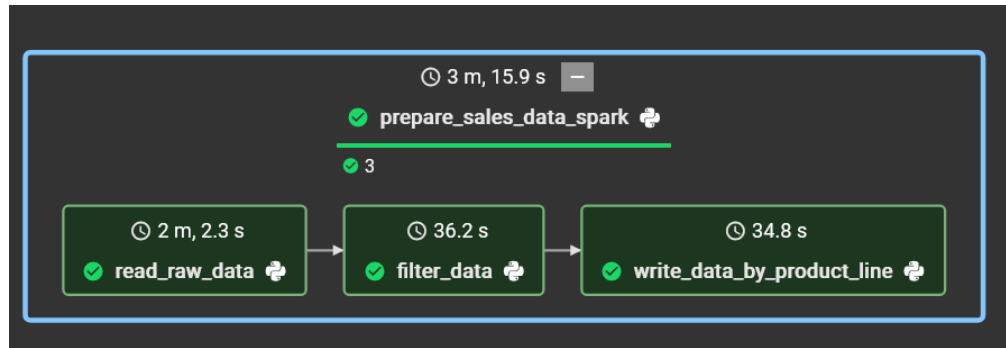


6. If you wish, introduce errors in the notebook.

# Part 3: Create pipelines with lineage

In this section you will review and run pipelines that create a lineage graph.

Lineage in Databand is created when the same dataset is used by multiple pipelines. Datasets are identified by the value that's provided to the logger function call.

For example, pipeline 1 writes the *Camping_Equipment.csv* file:

```
# Log writing the Camping Equipment csv
with dataset_op_logger("local://WeeklySales/Camping_Equipment.csv", "write", with_schema=True,
    logger.set(data=campingEquipment)

campingEquipment.to_csv("Camping_Equipment.csv", index=False)
```

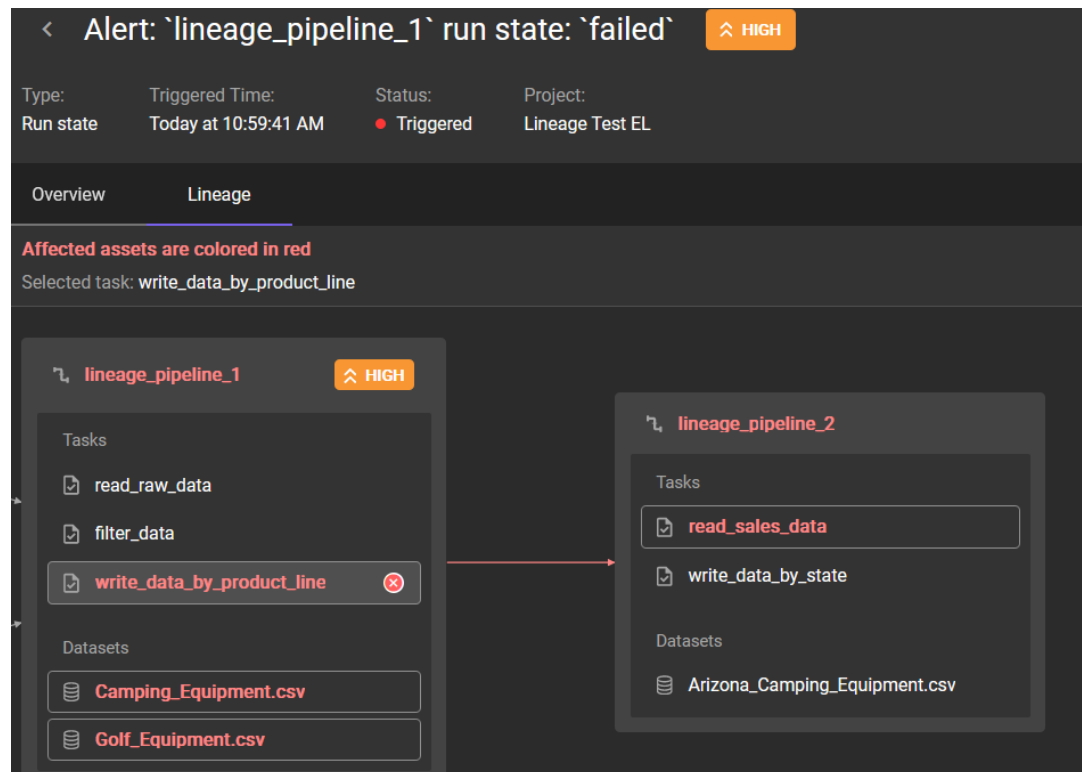Pipeline 2 reads the same file:

```
def read_sales_data():

    retailData = pd.read_csv('Camping_Equipment.csv')

    # Log the data read
    with dataset_op_logger("local://Weekly_Sales/Camping_Equipment.csv", "read",
        logger.set(data=retailData)

    return retailData
```

In Databand, the graphic lineage is represented like this:

Databand is able to create this lineage because we provided the same value, *local://Weekly_Sales/Camping_Equipment.csv* to the logger.

While this value can be *any string*, the recommendation is to make it as close to the actual source path as possible to simplify troubleshooting. We chose "local" to indicate that in our example we are reading from the local data source. If your data source is in DB2, you can include database name, schema, and table as a part of the string passed to the logger. For example "*db2://bludb/sales/camping_equipment*".

Unlike pipeline steps and dataset statistics, lineage is not determined at the time of a pipeline run. Lineage jobs run automatically every 4 hours, and read/write operations on datasets for the past 7 days are evaluated. At this time these settings are not configurable, however, they are displayed on the **Lineage** tab.
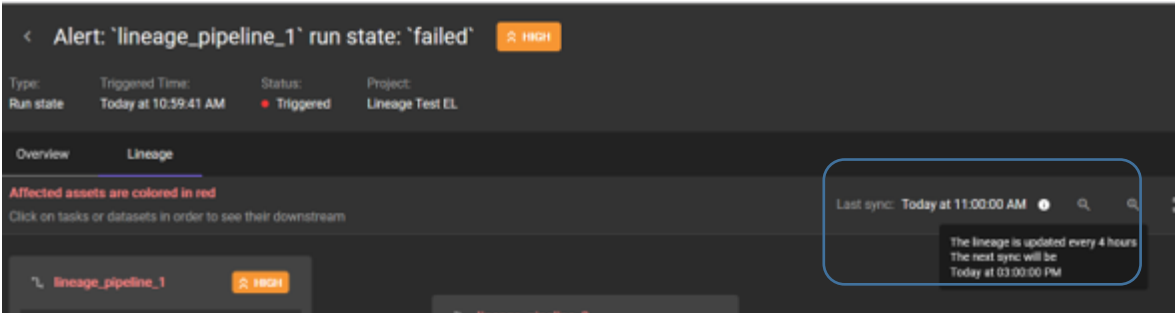


In this lab we will first review lineage for pipelines that already ran. Then you will run your pipelines, however the complete lineage will not be available at least 2 hours after the run.

1. In **Alerts** page filter by project *Lineage Test* and find an alert for *lineage_pipeline_1* (failed).
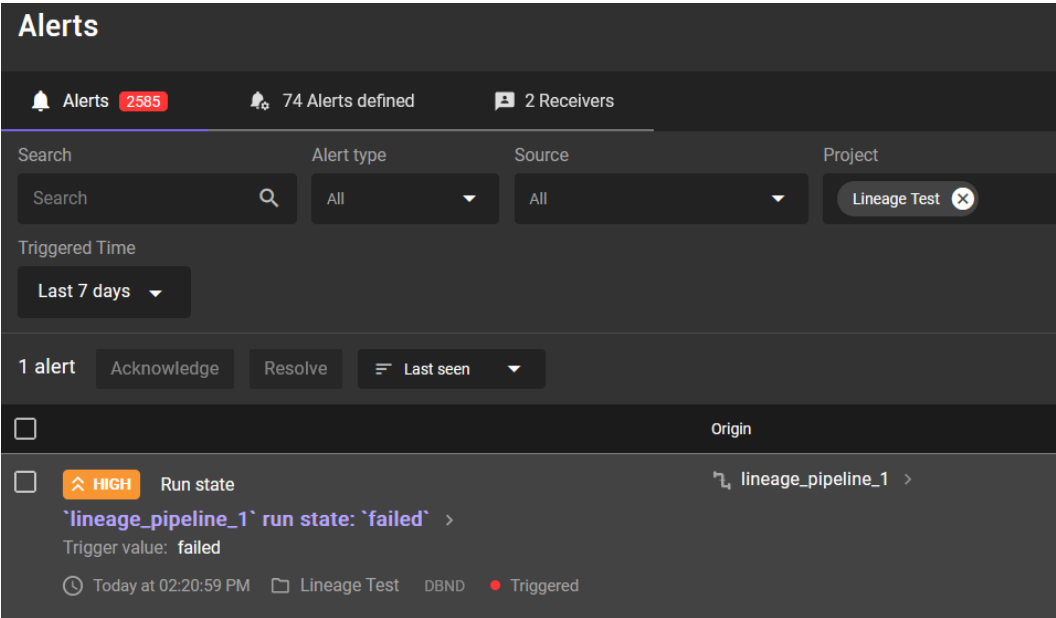
2.  Click on the alert, then on the **Lineage** tab.

    In the **Alerted pipeline** view we can see that the *write_data_by_product_line* step failed, and that failure affected writing two datasets – *Camping_Equipment* and *Golf_Equipment*.



Click on the **Downstream pipelines** tab. Here we can see that the issue in *lineage_pipeline1* affected *lineage_pipeline2*. As you click on different Tasks (steps) in *lineage_pipeline1*, the affect tasks and datasets are highlighted in boxes. Items in red font were directly affected by the failed pipeline run.
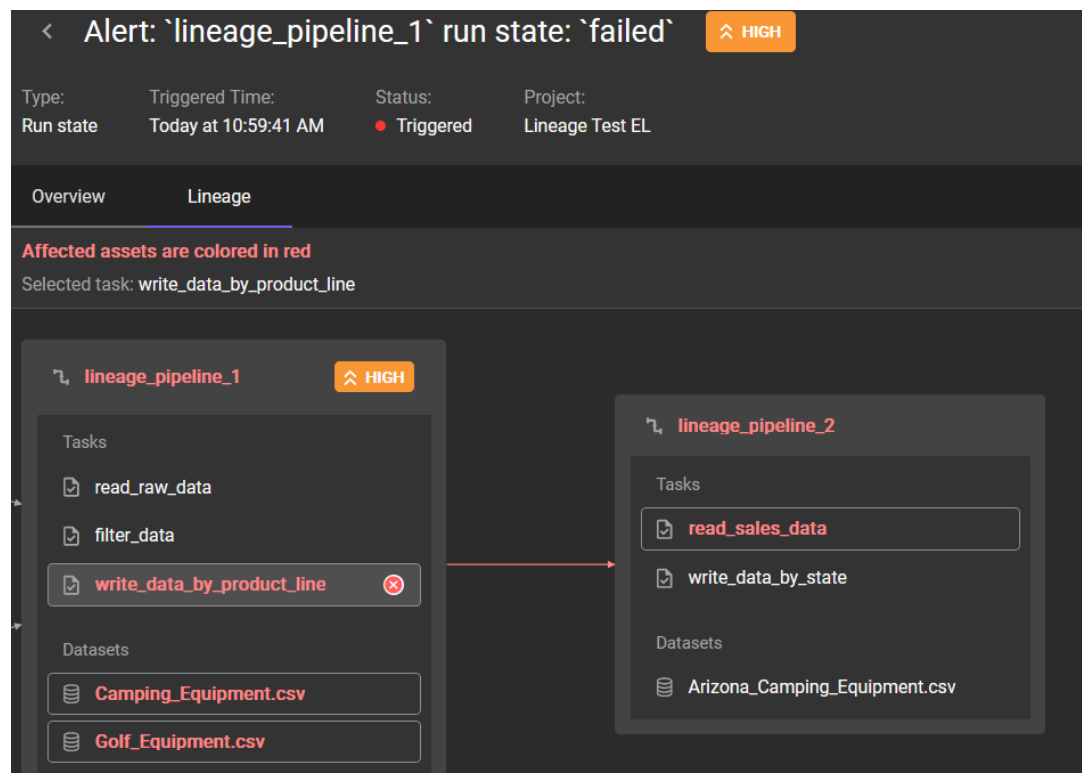
Next, you will recreate the same lineage by running your pipelines.

3. Load *Lineage_Pipeline1* and *Lineage_Pipeline2* Python scripts or notebooks to your Python environment.

4. In both scripts

- Replace the *url* and *token* variables with the values for your Databand cluster.

```
# Import pandas and databand libraries
import pandas as pd
from dbnd import dbnd_tracking, task, dataset_op_logger

databand_url = 'insert_url'
databand_access_token = 'insert_token'
```

- Replace the value of the *unique_suffix* variable to your initials.

```
# Provide a unique suffix that will be added to various assets tracked in Databand. We use this approach because
# in a workshop many users are running the same sample pipelines
unique_suffix = '_el'
```

5. Run *Lineage_Pipeline1* 3-5 time, then *Lineage_Pipeline2 (*also 3-5 times).

Currently the Lineage graph is accessible through the **Alerts** page only, which means that we have to create an Alert for *Lineage_Pipeline1*.

While you can create an alert for a successful run, in most cases data engineers want to investigate lineage for failed runs. We recommend that you "force" an error in the last step of *Lineage_Pipeline1.*

6. In Databand **Pipelines** tab find your pipeline and create an alert for pipeline status *failed*.
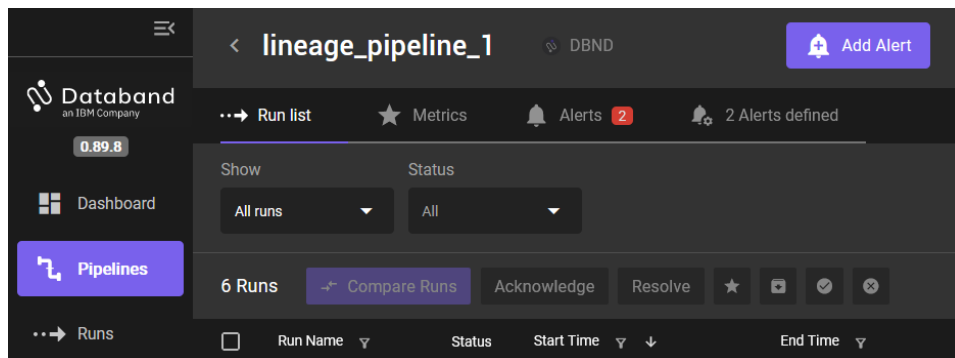
7. We will use the same approach as in **Part 1** to force an error in *LineagePipeline1* – we will change the code to the name of the dataframe that does not exist (*golfEquipment1*) in *write_data_by_product_line* function.

```
# Select any product line
golfEquipment = filteredData.loc[filteredData['Product line'] == 'Golf Equipment']

# Log the filtered data read
with dataset_op_logger(unique_file_name_2, "write", with_schema=True,
                        with_preview=True) as logger:
    # Write the csv file
    golfEquipment1.to_csv("Golf_Equipment.csv", index=False)

    logger.set(data=golfEquipment)
```

8. Run the modified pipeline to generate an alert.

   Drill down to the alert and notice that we have 2 tabs under **Impact Analysis**. If you click on the **Lineage** tab, you will see the **Affected pipeline** lineage, but the **Downstream lineage** will display a *"Can't track lineage"* message*.*

**Downstream pipelines** as well as **Affected pipelines tab** information will be available after the automatic lineage jobs run (typically every 4 hours).

You can check your existing alerts after a few hours to check if they have been updated. In the **Impact Analysis** section you will see the **Affected Pipelines** tab.
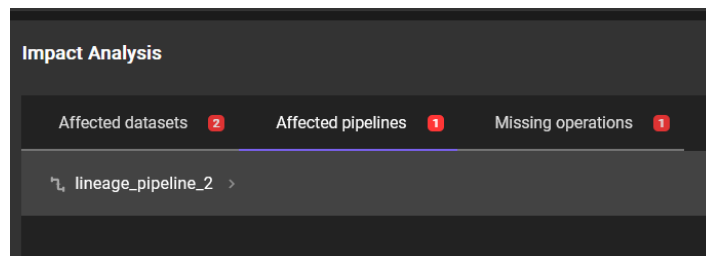


**Downstream pipelines** output:



You have completed the Lineage section of the lab.

# Part 4: Monitoring other pipelines

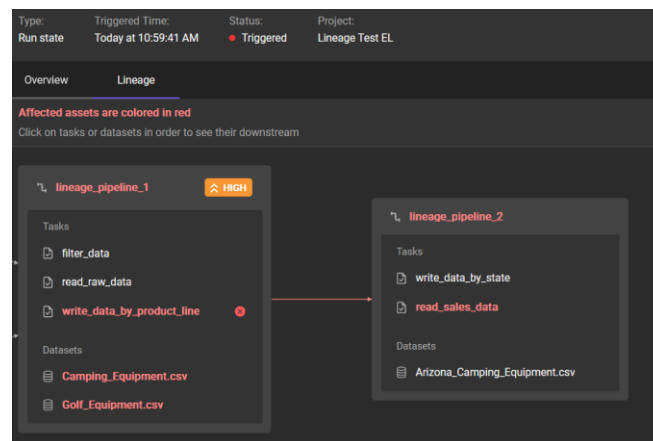While Databand is primarily used to monitor data pipelines, the SDK can also be used to monitor Machine Learning pipelines.

In addition to monitoring pipeline status, logging datasets that are used for model building and scoring, as well as capturing some statistics for models (such as feature importance or model accuracy) can be useful for creating different types of alerts. For example:

- Schema change in a model building pipeline
- Schema change in a model scoring pipeline
- Change in feature importance
- Values below threshold in a model building pipeline.

We created a sample notebook that demonstrates how the Databand SDK can be used in a model building pipeline.

In this section you will review the notebook.

1. Log in to **Cloud Pak for Data as a Service** (**CPDaaS**). Create a new project or open an existing project.

2. Create a notebook from file: *Model_build_with_Databand* (in the */Pipelines* folder downloaded from Box).

   We do not need to import data because it's read from a Git repo.

3. Review the notebook and make the changes as provided in the notebook instructions.

   To understand the changes that were made for tracking, look for the # DATABAND tag

   ```
   # DATABAND
   # Run once during notebook execution to install the Databand SDK
   !pip install databand
   ```

   ```
   # DATABAND
   # Import Databand libraries
   from dbnd import dbnd_tracking, task, dataset_op_logger,log_metric
   ```

4. Run the notebook and check results in Databand.

5. If you wish, set up alerts for model accuracy threshold (make it less than 0.75).

   You have completed this section of the lab.