



Data Sheet

Data Quality in Databricks with Great Expectations (GX Library)

5201 GREAT AMERICAN PARKWAY, SUITE 320
SANTA CLARA, CA 95054

Tel: (855) 695-8636

E-mail: info@lumendata.com

Website: www.lumendata.com

Great Expectations (GE/GX) is a comprehensive Python library specifically designed to address the critical task of data quality management.

Core functionalities:

- **Expectation Definition:** Through a flexible and extensible framework, GX enables the creation of human-readable expectations that explicitly define the intended characteristics of your data. This encompasses aspects such as value ranges, data types, completeness, and even complex patterns.
- **Validation and Monitoring:** Leveraging these defined expectations, GX acts as a powerful validation engine. It meticulously scans your data lakehouse and identifies any deviations or anomalies that breach your established quality standards.
- **Comprehensive Documentation:** GX meticulously documents both the defined expectations and the subsequent validation outcomes.
- **Integration and Automation:** Built for seamless integration with popular tools like Databricks, GX effortlessly embeds data validation within your existing data pipelines. Moreover, you can schedule and automate these checks, enabling continuous monitoring and proactive identification of potential quality issues.

By effectively utilizing GX, you can establish a rigorous data quality culture within your organization. This ensures the consistency, accuracy, and trustworthiness of your data, ultimately resulting in reliable insights, informed decision-making, and a robust data-driven environment.

[Documentation:](#)

Prerequisites

1. A complete Databricks setup, including a running Databricks cluster with an attached notebook
2. Access to DBFS

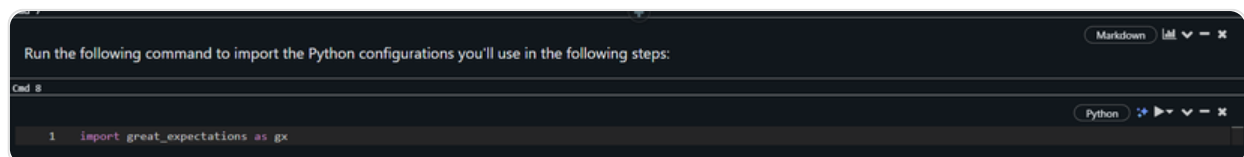
Install GX Dependencies

Run the following command in your notebook to install GX as a notebook-scoped library:



```
Cell 5
Install GX Dependencies
Run the following command in your notebook to install GX as a notebook-scoped library:
Cell 6
!pip install great-expectations
Requirement already satisfied: beautifulsoup4 in /databricks/python3/11h/python3.10/site-packages (from nbconvert>=5.7.0; notebook>=6.4.0; great-expectations) (4.11.1)
```

Run the following command to import the Python configurations you'll use in the following steps:



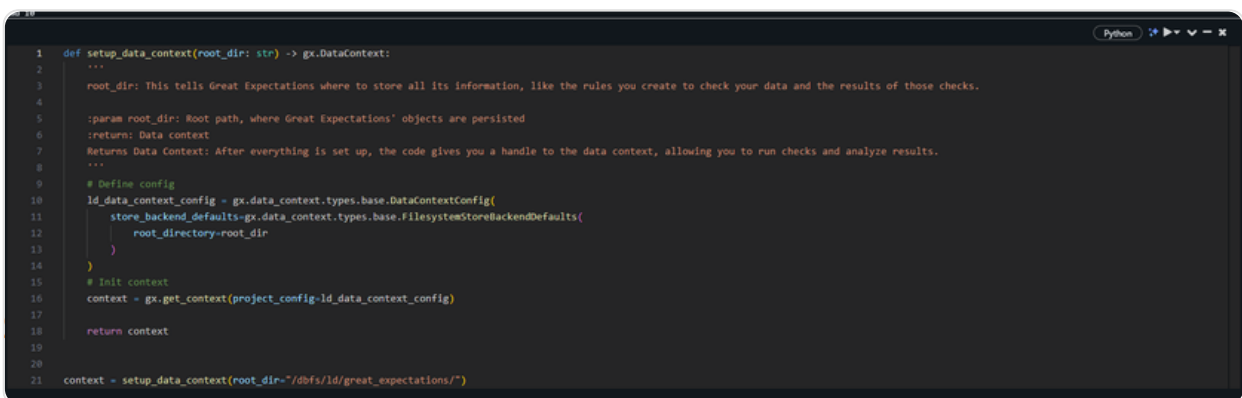
```
Cell 8
Run the following command to import the Python configurations you'll use in the following steps:
1 import great_expectations as gx
```

Setting up GX Data Context

1. A Data Context is the primary entry point for a Great Expectations (GX) deployment, and it provides the configurations and methods for all supporting GX components.
2. A Data Context also allows you to configure top-level components, and you can use different storage methodologies to back up your Data Context configuration.
3. This code snippet shows how to use Great Expectations within Databricks, storing everything on the Databricks File System (DBFS). This makes sure your data quality checks and results are accessible and shareable across different Databricks jobs.

Here's what's happening:

1. **Data Context Setup:** The code creates a data context, which is like a central hub for all your Great Expectations tasks. It tells GX where to find your data and where to store its findings.
2. **DBFS Storage:** We choose DBFS as the storage location, so everything stays within the Databricks environment. Think of DBFS as a shared folder accessible to all your Databricks jobs.
3. **Persistence and Sharing:** By using DBFS, your expectations (rules for your data) and validation results are saved for everyone to see and use. This promotes collaboration and ensures everyone uses the same data checks.



```
1 def setup_data_context(root_dir: str) -> gx.DataContext:
2     """
3     root_dir: This tells Great Expectations where to store all its information, like the rules you create to check your data and the results of those checks.
4
5     :param root_dir: Root path, where Great Expectations' objects are persisted
6     :return: Data context
7     Returns Data Context: After everything is set up, the code gives you a handle to the data context, allowing you to run checks and analyze results.
8     """
9
10    # Define config
11    ld_data_context_config = gx.data_context.types.base.DataContextConfig(
12        store_backend_defaults=gx.data_context.types.base.FilesystemStoreBackendDefaults(
13            root_directory=root_dir
14        )
15    )
16    # Init context
17    context = gx.get_context(project_config=ld_data_context_config)
18    return context
19
20
21 context = setup_data_context(root_dir="/dbfs/ld/great_expectations/")
```

Connecting Data Source to Great Expectations in Databricks

After setting up your data Context, you need to tell Great Expectations where to find your data. This is where datasources come in. Think of them as bridges connecting your data to your checks.

This code example creates a datasource called `spark_data_source` specifically for Spark dataframes.

Here's the breakdown:

1. **Execution Engine:** Like a powerful analyzer, this engine checks your data for quality (SparkDFExecutionEngine in this case).
2. **Data Connector:** This "bridge" connects to your specific data source (e.g., a file or database). We'll use a basic one for now.
3. Every data connector defines batch identifiers. Those will be specified for every batch request to identify the batch.

```
1 def setup_spark_data_source(context: gx.DataContext):  
2     ...  
3     Spark Datasource: It creates a special bridge called spark_data_source for your Spark data.  
4     Spark Connector: This bridge (called spark_data_source_connector) tells Great Expectations how to access your data within Spark.  
5     Timestamp Identifier: To track when data changes, it uses "timestamp" as a unique label for each batch of data.  
6  
7     :param context: Data context  
8     ...  
9     source = gx.datasource.Datasource(  
10         name="spark_data_source",  
11         execution_engine={  
12             "module_name": "great_expectations.execution_engine",  
13             "class_name": "SparkDFExecutionEngine"  
14         },  
15         data_connectors={  
16             f"spark_data_source_connector": {  
17                 "class_name": "RuntimeDataConnector",  
18                 "batch_identifiers": ["timestamp"]  
19             }  
20         }  
21     )  
22     context.add_or_update_datasource(datasource=source)  
23  
24  
25  
26 setup_spark_data_source(context)
```

Getting Your Data Ready for Quality Checks by Setting Up Runtime Batch Request

Now that Great Expectations is connected to our Spark data, we need to fetch/get only specific portion of the data. This is where batch requests come in.

Think of a batch request as a specific slice of your data:

- **Batch Request Creation:** It creates a request to analyze a batch of data from your Spark dataframe.
- **Caching for Speed:** To make things faster, it stores (caches) this data chunk temporarily.
- **New York Taxi Data:** For this example, it uses the readily available New York taxi dataset in Databricks.

```
1 import pyspark as ps
2
3
4 def ld_create_batch_request(df: ps.sql.DataFrame, df_name: str, timestamp: str) -> gx.core.batch.RuntimeBatchRequest:
5     """
6     Creates a batch request from a Spark data frame.
7
8     :param df: Spark data frame
9     :param df_name: Name of the spark data frame e.g. table name
10    :param timestamp: Timestamp string to be passed as a batch identifier
11    :return: Batch request
12    """
13    runtime_batch_request = gx.core.batch.RuntimeBatchRequest(
14        datasource_name="spark_data_source",
15        data_connector_name="spark_data_source_connector",
16        data_asset_name="spark_batch_{}.{}".format(df_name, timestamp),
17        runtime_parameters={"batch_data": df},
18        batch_identifiers={
19            "timestamp": timestamp,
20        }
21    )
22    return runtime_batch_request
23
24
25 def current_timestamp() -> str:
26     """
27     Returns the current timestamp.
28
29     :return: Timestamp string in ISO 8601 format
30     """
31     import datetime
32     return datetime.datetime.utcnow().isoformat()
```

```
34
35 # Create Spark app
36 spark = ps.sql.SparkSession.builder.appName("myc_taxi_great_expectations_checks").getOrCreate()
37
38 # Cache table for faster runtime
39 #
40 # NOTE
41 # If there is a timestamp column, you may add a filter for a specific time frame.
42 # This allows to rerun the checks later using the timestamp from the batch.
43 #
44 table = "samples.nyctaxi.trips"
45 spark.sql(f"CACHE TABLE ld_data_nyc AS SELECT * FROM {table}")
46
47 # Get data frame
48 df = spark.sql("SELECT * FROM ld_data_nyc")
49
50 # Create batch request
51 timestamp = current_timestamp()
52 batch_request = ld_create_batch_request(df=df, df_name=table, timestamp=timestamp)
53
54 2) Spark Jobs
55 df: pyspark.sql.dataframe.DataFrame = [tpep_pickup_datetime: timestamp, tpep_dropoff_datetime: timestamp ... 4 more fields]
```

Building Your Data Quality Rules with Great Expectations Suite:

Expectation Creation Methods:

- **Automated Profiler:** This tool analyzes your data sample and generates expectations based on its characteristics. It establishes a baseline for data when you're new to Great Expectations.
- **Manual Creation:** Based on your domain knowledge, we can define fine grained specific expectations to address unique data qualities.

Using a profiler

The profiler is a helpful tool that analyzes data sample and automatically generates expectations based on its characteristics.

Generating Expectations from a Sample:

In the previous section, we created a batch request. Now, let's leverage the profiler to generate expectations from that sample:

- **Feeding the Data:** We'll use the batch request as input for the profiler.
Automatic Analysis: The profiler analyzes the sample, identifying patterns and characteristics.
- **Crafting Expectations:** Based on its analysis, the profiler creates expectations, like rules for your data.

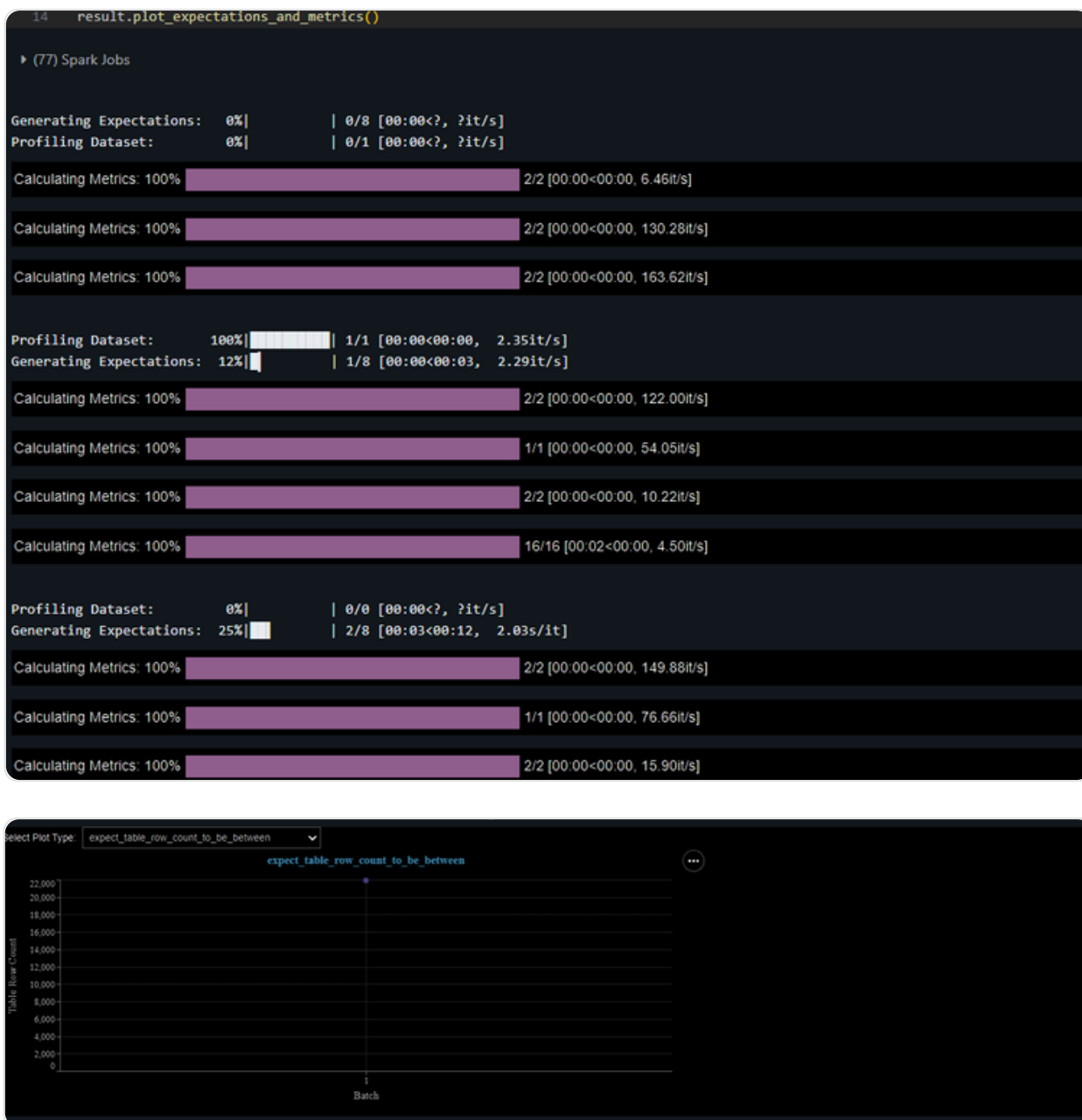
Benefits of Automated Expectations:

Quick Start: The profiler offers a time-saving way to establish basic expectations.

Initial Insights: It provides an early understanding of your data's quality landscape

The code snippet below demonstrates how to leverage the profiler. It generates expectations automatically and showcases the results.

```
1  # Run the default onboarding profiler on the batch request
2  result = context.assistants.onboarding.run(
3      batch_request=batch_request,
4      exclude_column_names=[],
5  )
6
7  # Get the suite with specific name
8  suite_name = "ld_nyc_taxi_expectations_v1"
9  suite = result.get_expectation_suite(
10     expectation_suite_name=suite_name
11 )
12
13 # Plot results
14 result.plot_expectations_and_metrics()
```

Fine-Tuning with Manual Expectations

While the profiler offers a convenient starting point, sometimes more control is needed. That's where we can leverage manual expectation creation.

Benefits of Manual Approach:

- 1. Specific Requirements:** Leverage your specific knowledge about your data to define specific checks that address unique qualities.
- 2. Granular Control:** Define expectations with more in-depth details.

Exploring the Expectations Gallery:

This online resource, here, serves as your library of pre-defined expectations. Browse through various categories to find ready-made checks.

The New York Taxi Example:

Added an expectation to our existing suite (from the profiler), ensuring that in our New York taxi trip data, dropoff times always occur later than pickup times.

Breakdown:

1.Suite Selection: We target the specific expectation suite generated earlier, where we want to add our new rule.

2.Expectation Definition: We define a clear expectation: dropoff_time > pickup_time. This ensures dropoff always occurs after pickup.

3.Validation Integration: We integrate this expectation into the validation process, ensuring it's included in future data checks.



```
1 # Create expectation config
2 config = gx.core.ExpectationConfiguration(
3     expectation_type="expect_column_pair_values_a_to_be_greater_than_b",
4     kwargs={
5         "column_A": "tpep_dropoff_datetime",
6         "column_B": "tpep_pickup_datetime"
7     },
8     meta={
9         "notes": {
10             "format": "markdown",
11             "content": "Dropoff time must be 'larger than' pickup time "
12         }
13     }
14 )
15
16 # Add expectation to suite
17 suite.add_expectation(expectation_configuration=config)
```

["expectation_type": "expect_column_pair_values_a_to_be_greater_than_b", "kwargs": {"column_A": "tpep_dropoff_datetime", "column_B": "tpep_pickup_datetime"}, "meta": {"notes": {"format": "markdown", "content": "Dropoff time must be 'larger than' pickup time "}}]

Securing Your Quality Checks: Saving Your Expectation Suite

Benefits:

Persistent Quality Rules: Saving ensures your expectations are readily available for future data checks, maintaining consistent quality standards.

Collaboration: Share your suite with others, promoting a unified understanding of data quality requirements across your team.

Version Control: Track changes and manage different versions of your expectations over time.

```
1 # Persist expectation suite with the specified suite name from above (suite_name="ld_nyc_taxi_expectations_v1")
2 context.add_or_update_expectation_suite(expectation_suite=suite)

{
  "column": "tpep_pickup_datetime",
  "meta": {
    "notes": {
      "format": "markdown",
      "content": "Dropoff time must be 'larger than' pickup time "
    }
  }
},
"data_asset_type": null,
"meta": {
  "great_expectations_version": "0.18.8",
  "citations": [
    {
      "citation_date": "2024-02-12T07:16:39.178291Z",
      "comment": "Created by effective Rule-Based Profiler of OnboardingDataAssistant with the configuration included.\n"
    }
  ]
}
}
```

Validate Your Data: Utilizing Great Expectations' Checkpoints

Building your expectation suite establishes your quality standards. Checkpoints in Great Expectations serve as the workhorses of production-level data quality assurance.

Key Features:

- **Bundled Validation:** Checkpoints can validate multiple data batches against a single expectation suite, streamlining the process.
- **Actionable Insights:** They allow you to specify actions triggered by validation results, such as sending alerts upon failure.

We can configure checkpoints to trigger various actions, including:

- **Alerts:** Receive notifications via email, Slack, or other channels when issues arise, ensuring data quality concerns.

By leveraging checkpoints effectively, you can:

- **Guarantee Data Reliability:** Maintain the accuracy and integrity of your data platform.
- **Proactive Issue Management:** React to potential quality problems, preventing downstream impact and ensuring timely resolution.
- **Streamlined Validation:** Bundle multiple data checks and actions, simplifying your quality assurance process.

The following code demonstrates how to create a checkpoint in Great Expectations:

Cmd 24

```
1 checkpoint_name="ld_nyc_taxi_test_checkpoint"
2
3 # Create and persist checkpoint to reuse for multiple batches
4 context.add_or_update_checkpoint(
5     name = checkpoint_name,
6     config_version = 1,
7     class_name = "SimpleCheckpoint",
8     validations = [
9         {"expectation_suite_name": suite_name}
10    ]
11 )
```


```
{
  "name": "update_data_docs",
  "action": {
    "class_name": "UpdateDataDocsAction"
  }
},
"batch_request": {},
"class_name": "SimpleCheckpoint",
"config_version": 1.0,
"evaluation_parameters": {},
"module_name": "great_expectations.checkpoint",
"name": "ld_nyc_taxi_test_checkpoint",
"profilers": [],
"runtime_configuration": {},
"validations": [
  {
    "expectation_suite_name": "ld_nyc_taxi_expectations_v1"
  }
]
```

Initiating Data Validation: Executing Checkpoints.

This section guides you through the process of running checkpoints within your Great Expectations environment.

Key Steps:

- **Checkpoint Selection:** Specify the exact checkpoint to execute, activating its validation process.
- **Batch Request Creation (Optional):** If checkpoint doesn't have predefined data batches, we can create one within the same notebook, ensuring we target the specific data segment.
- **Checkpoint Execution:** Run the chosen checkpoint, triggering the validation process against expectations.



```
1 # Run checkpoint
2 checkpoint_result = context.run_checkpoint(
3     checkpoint_name=checkpoint_name,
4     batch_request=batch_request
5 )
```

► (44) Spark Jobs

Calculating Metrics: 100% 130/130 [00:09<00:00, 15.70it/s]

Visualizing Checkpoint Results

After running a checkpoint, Great Expectations automatically generates data docs. These are informative reports presenting validation results in a clear and visually appealing format.

Benefits of Data Docs:

- **Easy Interpretation:** Ditching raw data, data docs translate results into an intuitive HTML format, making quality insights readily accessible.
- **Databricks Integration:** Seamlessly view data docs right within the Databricks environment using the `displayHTML` call.

The **provided code snippet** demonstrates how to leverage `displayHTML` to visualize the checkpoint results from the previous section. This lets you analyze the validation outcome directly within your Databricks notebook.

What to expect from generated Data Docs:

- **Overall Validation Status:** Was the data compliant with your expectations?
- **Detailed Expectation Results:** Dive deeper into individual expectations to understand specific successes and failures.
- **Visualization and Metrics:** Charts and graphs provide a clear picture of data quality trends and potential issues.

Data Quality in Databricks with Great Expectations

```
1 def display_checkpoint_results(context: gx.DataContext, result: gx.checkpoint.types.checkpoint_result.CheckpointResult):
2     ...
3     Displays the docs generated by performing a checkpoint in a Databricks notebook.
4
5     :param context: Data context
6     :param result: Checkpoint result
7     ...
8     result_ids = result.list_validation_result_identifiers()
9     # Iterate validations results
10    for result_id in result_ids:
11        docs = context.get_docs_sites_urls(resource_identifier=result_id)
12        # Iterate and display docs
13        for doc in docs:
14            path = doc["site_url"]
15            # Remove the file:// prefix from the URL
16            if path.startswith("file:///"):
17                path = path[len("file:///"): ]
18            # Display the HTML in the Databricks notebook
19            with open(path, "r") as f:
20                displayHTML(f.read())
21
22    display_checkpoint_results(context, checkpoint_result)
```

great expectations

Home / Validations / ld_nyc_taxi_expectations_v1 / spark_batch_samples.nyc taxi.trips_2024-02-12T06:52:38.185213 / __none__ / 2024-02-12T08:20:40Z

Expectation Validation Result

Evaluates whether a batch of data matches expectations.

Actions

Validation Filter:

Show All Failed Only

How to Edit This Suite

Show Walkthrough

Table of Contents

Overview

Overview

Expectation Suite: ld_nyc_taxi_expectations_v1
Data asset: spark_batch_samples.nyc taxi.trips_2024-02-12T06:52:38.185213
Status: ✗ Failed

Statistics

Evaluated Expectations	43
Successful Expectations	42
Unsuccessful Expectations	1
Success Percent	≈97.67%

Show more info...

Table-Level Expectations

Status	Expectation	Observed Value
✓	0 Must have greater than or equal to 21932 and less than or equal to 21932	21932

Table of Contents

Overview

Table-Level Expectations

dropoff_zip

fare_amount

pickup_zip

tpep_dropoff_datetime

tpep_pickup_datetime

trip_distance

Status **Expectation** **Observed Value**

✓	0 Must have greater than or equal to 21932 and less than or equal to 21932 rows.	21932
✓	0 Must have at least these columns (in any order): trip_distance, dropoff_zip, tpep_pickup_datetime, tpep_dropoff_datetime, pickup_zip, fare_amount	['tpep_pickup_datetime', 'tpep_dropoff_datetime', 'trip_distance', 'fare_amount', 'pickup_zip', 'dropoff_zip']
✗	0 Values in tpep_dropoff_datetime must always be greater than those in tpep_pickup_datetime. 1 unexpected values found. ≈0.00456% of 21932 total rows.	≈0.0045595% unexpected
	Unexpected Value Count 2016-01-28T16:02:422016-01-28T16:02:42 1	

dropoff_zip

Status	Expectation	Observed Value
✓	0 values must never be null.	100% not null
✓	0 minimum value must be greater than or equal to 7002 and less than or equal to 7002.	7002
✓	0 maximum value must be greater than or equal to 11694 and less than or equal to 11694.	11694

trip_distance

Search

Status	Expectation	Observed Value																				
✓	0 values must never be null.	100% not null																				
✓	0 minimum value must be greater than or equal to 0.0 and less than or equal to 0.0.	0																				
✓	0 maximum value must be greater than or equal to 30.6 and less than or equal to 30.6.	30.6																				
✓	0 values must be greater than or equal to 0.0 and less than or equal to 30.6.	0% unexpected																				
✓	0 quantiles must be within the following value ranges.																					
	<table><thead><tr><th>Quantile</th><th>Min Value</th><th>Max Value</th></tr></thead><tbody><tr><td>Q1</td><td>1.0</td><td>1.0</td></tr><tr><td>Median</td><td>1.69</td><td>1.69</td></tr><tr><td>Q3</td><td>3.08</td><td>3.08</td></tr></tbody></table>	Quantile	Min Value	Max Value	Q1	1.0	1.0	Median	1.69	1.69	Q3	3.08	3.08	<table><thead><tr><th>Quantile</th><th>Value</th></tr></thead><tbody><tr><td>Q1</td><td>1.0</td></tr><tr><td>Median</td><td>1.69</td></tr><tr><td>Q3</td><td>3.08</td></tr></tbody></table>	Quantile	Value	Q1	1.0	Median	1.69	Q3	3.08
	Quantile	Min Value	Max Value																			
	Q1	1.0	1.0																			
	Median	1.69	1.69																			
Q3	3.08	3.08																				
Quantile	Value																					
Q1	1.0																					
Median	1.69																					
Q3	3.08																					
✓	0 median must be greater than or equal to 1.69 and less than or equal to 1.69.	1.69																				
✓	0 mean must be greater than or equal to 2.8528291993434225 and less than or equal to 2.8528291993434225.	≈2.852829199																				
✓	0 standard deviation must be greater than or equal to 3.4399536210623523 and less than or equal to 3.4399536210623523.	≈3.439953621																				

Conclusion: Empowering Data Quality with Great Expectations and Databricks

This data sheet has explored how to leverage Great Expectations within Databricks to ensure the integrity and reliability of data.

By combining the power of Great Expectations with the flexibility of Databricks, we can create a robust data quality management system tailored to our needs.

We've provided a step-by-step guide, covering:

- Setting up Great Expectations in Databricks: Learn how to create a data context, connect to data sources, and define expectations.
- Executing Data Validation: Discover how to run checkpoints and schedule regular checks to proactively monitor your data quality.
- Visualizing Results: See how data docs offer clear insights into your validation outcomes, helping you understand and address any issues.

By following these steps, we can build a data quality system that supports our organization's data-driven goals. With consistent quality checks in place, we can make informed decisions, drive innovation, and confidently rely on data.

Reference links:

- https://docs.greatexpectations.io/docs/oss/get_started/get_started_with_gx_and_databricks/
- <https://greatexpectations.io/>

Authors



Ritesh Chidrewar

Senior Consultant - Data engineering

About LumenData

LumenData is a leading provider of **Enterprise Data Management, Cloud & Analytics** solutions. We help businesses navigate their data visualization and analytics anxieties and enable them to accelerate their innovation journeys.

Founded in 2008, with locations in multiple countries, LumenData is privileged to serve over 100 leading companies. LumenData is **SOC2 certified** and has instituted extensive controls to protect client data, including adherence to GDPR and CCPA regulations.



Get in touch with us:
info@lumendata.com

Let us know what you need:
lumendata.com/contact-us

