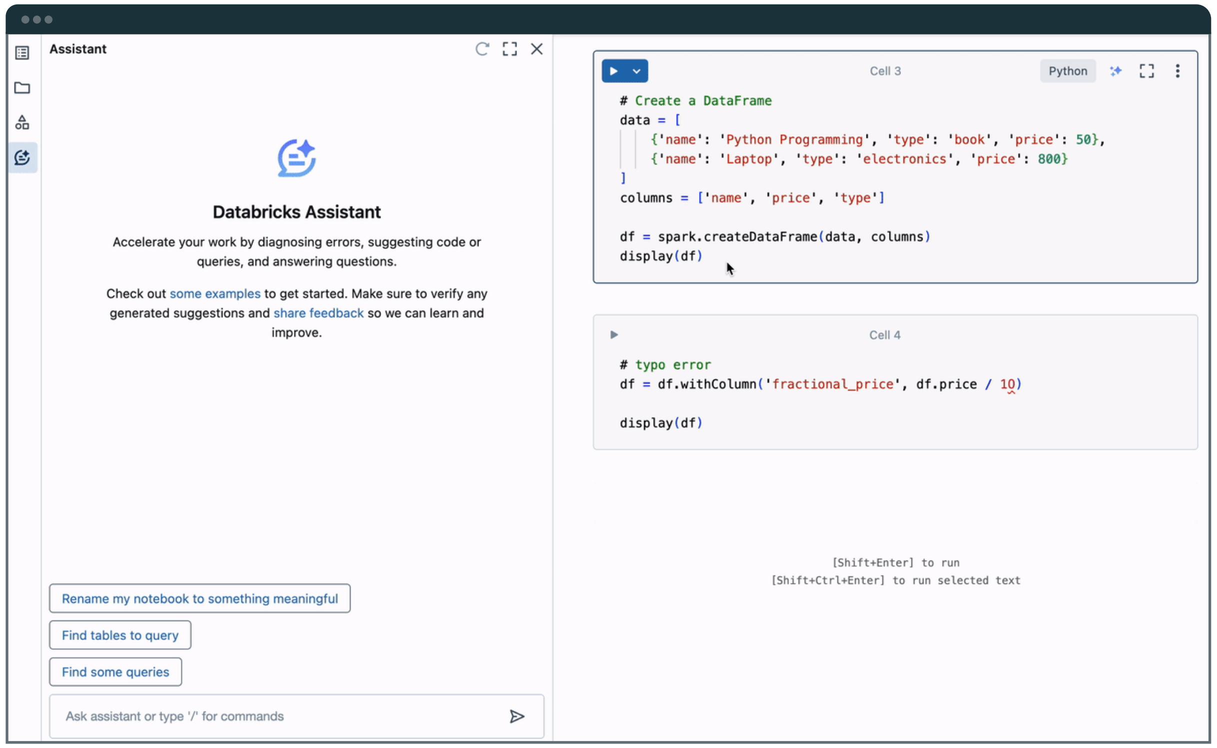
Diagnosing errors

Inevitably, data engineers will need to debug. The Assistant eliminates the need to open multiple browser tabs or switch contexts in order to identify the cause of errors in code, and staying focused is a tremendous productivity boost. To understand how this works with the Assistant, let's create a simple PySpark DataFrame and trigger an error.


Assistant
Cell 3
Python
Cell 4
Rename my notebook to something meaningful
Find tables to query
Find some queries
Ask assistant or type '/' for commands

Databricks Assistant

Accelerate your work by diagnosing errors, suggesting code or queries, and answering questions.

Check out some examples to get started. Make sure to verify any generated suggestions and share feedback so we can learn and improve.

```python
# Create a DataFrame
data = [
    {'name': 'Python Programming', 'type': 'book', 'price': 50},
    {'name': 'Laptop', 'type': 'electronics', 'price': 800}
]
columns = ['name', 'price', 'type']

df = spark.createDataFrame(data, columns)
display(df)
```

```python
# typo error
df = df.withColumn('fractional_price', df.price / 10)

display(df)
```

```
[Shift+Enter] to run
[Shift+Ctrl+Enter] to run selected text
```

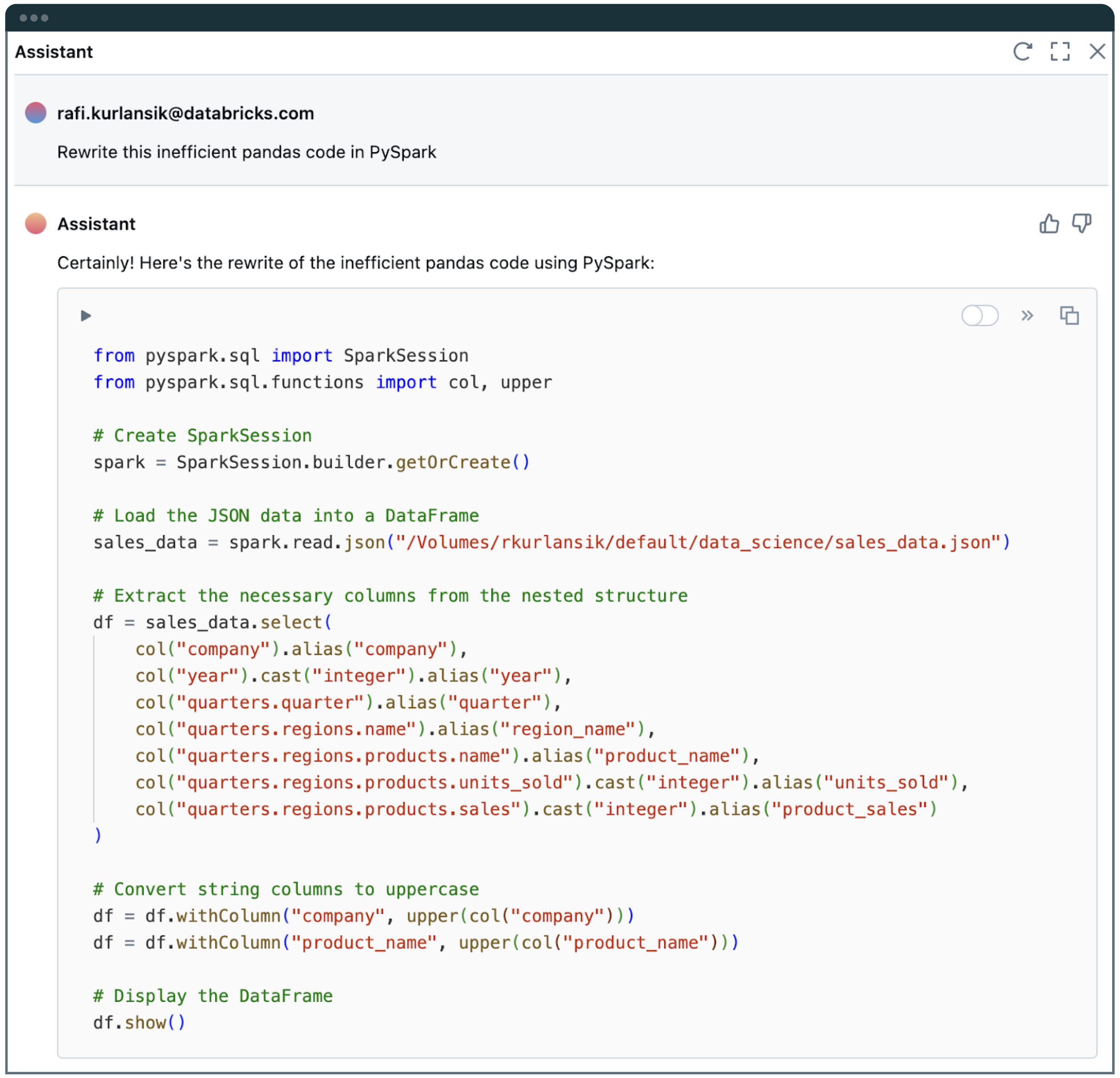
In the above example, a typo is introduced when adding a new column to the DataFrame. The zero in "10" is actually the letter "O", leading to an *invalid decimal literal* syntax error. The Assistant immediately offers to diagnose the error. It correctly identifies the typo, and suggests corrected code that can be inserted into the editor in the current cell. Diagnosing and correcting errors this way can save hours of time spent debugging.

Transpiling pandas to PySpark

Pandas is one of the most successful data-wrangling libraries in Python and is used by data scientists everywhere. Sticking with our JSON sales data, let's imagine a situation where a novice data scientist has done their best to flatten the data using pandas. It isn't pretty, it doesn't follow best practices, but it produces the correct output:

```python
import pandas as pd
import json

with open("/Volumes/rkurlansik/default/data_science/sales_data.json") as file:
    data = json.load(file)

# Bad practice: Manually initializing an empty DataFrame and using a deeply nested
for-loop to populate it.
df = pd.DataFrame(columns=['company', 'year', 'quarter', 'region_name', 'product_
name', 'units_sold', 'product_sales'])

for quarter in data['quarters']:
    for region in quarter['regions']:
        for product in region['products']:
            df = df.append({
                'company': data['company'],
                'year': data['year'],
                'quarter': quarter['quarter'],
                'region_name': region['name'],
                'product_name': product['name'],
                'units_sold': product['units_sold'],
                'product_sales': product['sales']
            }, ignore_index=True)

# Inefficient conversion of columns after data has been appended
df['year'] = df['year'].astype(int)
df['units_sold'] = df['units_sold'].astype(int)
df['product_sales'] = df['product_sales'].astype(int)

# Mixing access styles and modifying the dataframe in-place in an inconsistent
manner
df['company'] = df.company.apply(lambda x: x.upper())
df['product_name'] = df['product_name'].str.upper()
```

By default, Pandas is limited to running on a single machine. The data engineer shouldn't put this code into production and run it on billions of rows of data until it is converted to PySpark. This conversion process includes ensuring the data engineer understands the code and rewrites it in a way that is maintainable, testable, and performant. The Assistant once again comes up with a better solution in seconds.



rafi.kurlansik@databricks.com

Rewrite this inefficient pandas code in PySpark

Assistant

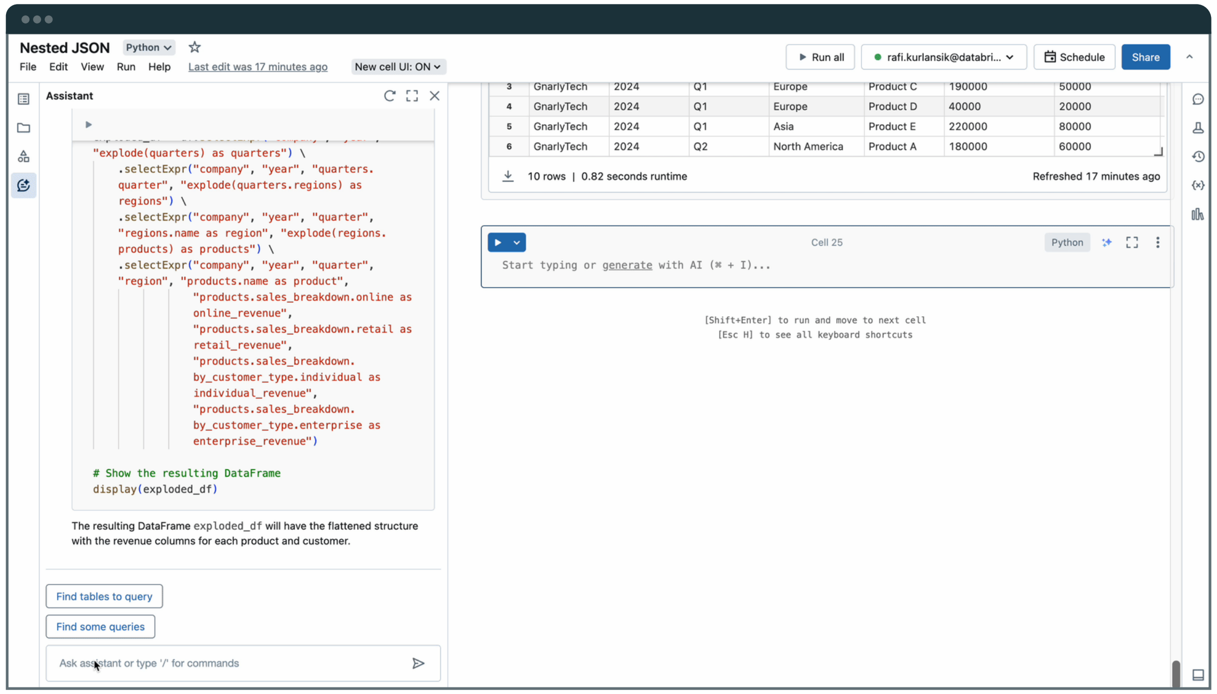
Certainly! Here's the rewrite of the inefficient pandas code using PySpark:

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, upper

# Create SparkSession
spark = SparkSession.builder.getOrCreate()

# Load the JSON data into a DataFrame
sales_data = spark.read.json("/Volumes/rkurlansik/default/data_science/sales_data.json")

# Extract the necessary columns from the nested structure
df = sales_data.select(
    col("company").alias("company"),
    col("year").cast("integer").alias("year"),
    col("quarters.quarter").alias("quarter"),
    col("quarters.regions.name").alias("region_name"),
    col("quarters.regions.products.name").alias("product_name"),
    col("quarters.regions.products.units_sold").cast("integer").alias("units_sold"),
    col("quarters.regions.products.sales").cast("integer").alias("product_sales")
)

# Convert string columns to uppercase
df = df.withColumn("company", upper(col("company")))
df = df.withColumn("product_name", upper(col("product_name")))

# Display the DataFrame
df.show()
```

Note the generated code includes creating a *SparkSession*, which isn't required in Databricks. Sometimes the Assistant, like any LLM, can be wrong or hallucinate. You, the data engineer, are the ultimate author of your code and it is important to review and understand any code generated before proceeding to the next task. If you notice this type of behavior, adjust your prompt accordingly.

WRITING TESTS

One of the most important steps in data engineering is to write tests to ensure your DataFrame transformation logic is correct, and to potentially catch any corrupted data flowing through your pipeline. Continuing with our example from the JSON sales data, the Assistant makes it a breeze to test if any of the revenue columns are negative – as long as values in the revenue columns are not less than zero, we can be confident that our data and transformations in this case are correct.



```python
"explode(quarters) as quarters") \
    .selectExpr("company", "year", "quarters.
    quarter", "explode(quarters.regions) as
    regions") \
    .selectExpr("company", "year", "quarter",
    "regions.name as region", "explode(regions.
    products) as products") \
    .selectExpr("company", "year", "quarter",
    "region", "products.name as product",
                "products.sales_breakdown.online as
                online_revenue",
                "products.sales_breakdown.retail as
                retail_revenue",
                "products.sales_breakdown.
                by_customer_type.individual as
                individual_revenue",
                "products.sales_breakdown.
                by_customer_type.enterprise as
                enterprise_revenue")

# Show the resulting DataFrame
display(exploded_df)
```

The resulting DataFrame `exploded_df` will have the flattened structure with the revenue columns for each product and customer.

3	GnarlyTech	2024	Q1	Europe	Product C	190000	50000
4	GnarlyTech	2024	Q1	Europe	Product D	40000	20000
5	GnarlyTech	2024	Q1	Asia	Product E	220000	80000
6	GnarlyTech	2024	Q2	North America	Product A	180000	60000

10 rows | 0.82 seconds runtime   Refreshed 17 minutes ago

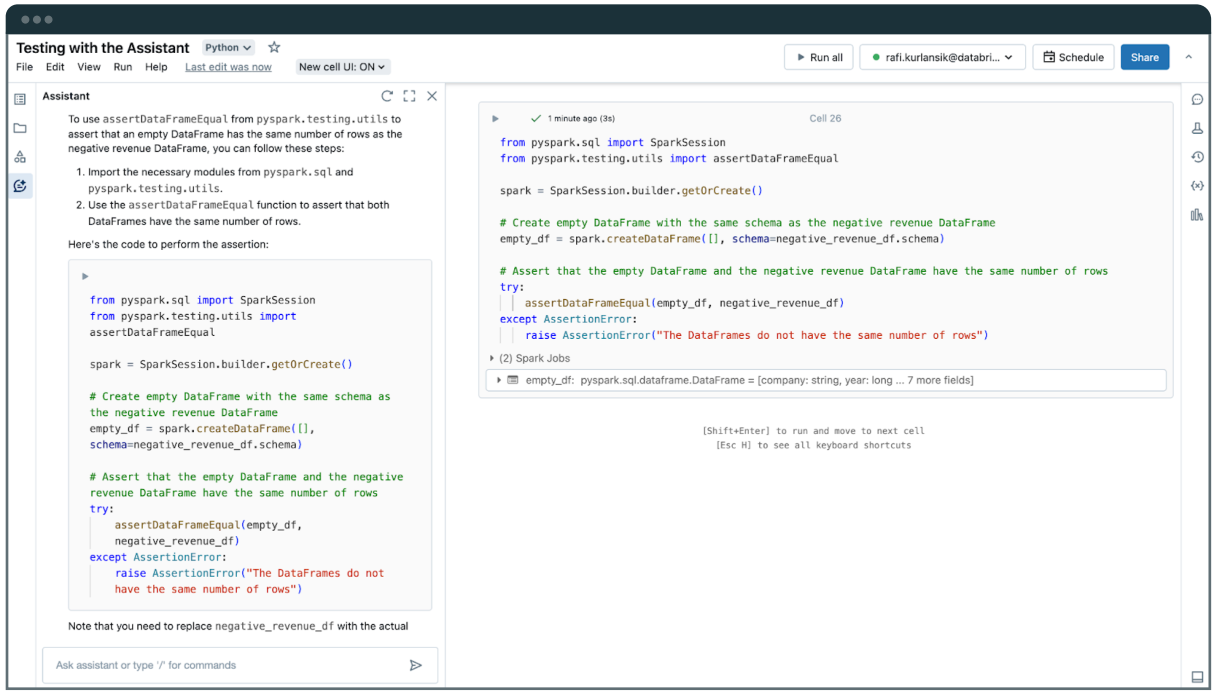
```
Start typing or generate with AI (⌘ + I)...

[Shift+Enter] to run and move to next cell
[Esc H] to see all keyboard shortcuts
```

Ask assistant or type '/' for commands

We can build off this logic by asking the Assistant to incorporate the test into PySpark's native testing functionality, using the following prompt:

Write a test using assertDataFrameEqual from pyspark.testing.utils to check that an empty DataFrame has the same number of rows as our negative revenue DataFrame.

The Assistant obliges, providing working code to bootstrap our testing efforts.



To use `assertDataFrameEqual` from `pyspark.testing.utils` to assert that an empty DataFrame has the same number of rows as the negative revenue DataFrame, you can follow these steps:

1. Import the necessary modules from `pyspark.sql` and `pyspark.testing.utils`.
2. Use the `assertDataFrameEqual` function to assert that both DataFrames have the same number of rows.

Here's the code to perform the assertion:

```
from pyspark.sql import SparkSession
from pyspark.testing.utils import
assertDataFrameEqual

spark = SparkSession.builder.getOrCreate()

# Create empty DataFrame with the same schema as
the negative revenue DataFrame
empty_df = spark.createDataFrame([],
schema=negative_revenue_df.schema)

# Assert that the empty DataFrame and the negative
revenue DataFrame have the same number of rows
try:
    assertDataFrameEqual(empty_df,
    negative_revenue_df)
except AssertionError:
    raise AssertionError("The DataFrames do not
    have the same number of rows")
```

Note that you need to replace `negative_revenue_df` with the actual

```
from pyspark.sql import SparkSession
from pyspark.testing.utils import assertDataFrameEqual

spark = SparkSession.builder.getOrCreate()

# Create empty DataFrame with the same schema as the negative revenue DataFrame
empty_df = spark.createDataFrame([], schema=negative_revenue_df.schema)

# Assert that the empty DataFrame and the negative revenue DataFrame have the same number of rows
try:
    assertDataFrameEqual(empty_df, negative_revenue_df)
except AssertionError:
    raise AssertionError("The DataFrames do not have the same number of rows")
```

```
(2) Spark Jobs
empty_df:  pyspark.sql.dataframe.DataFrame = [company: string, year: long ... 7 more fields]
```

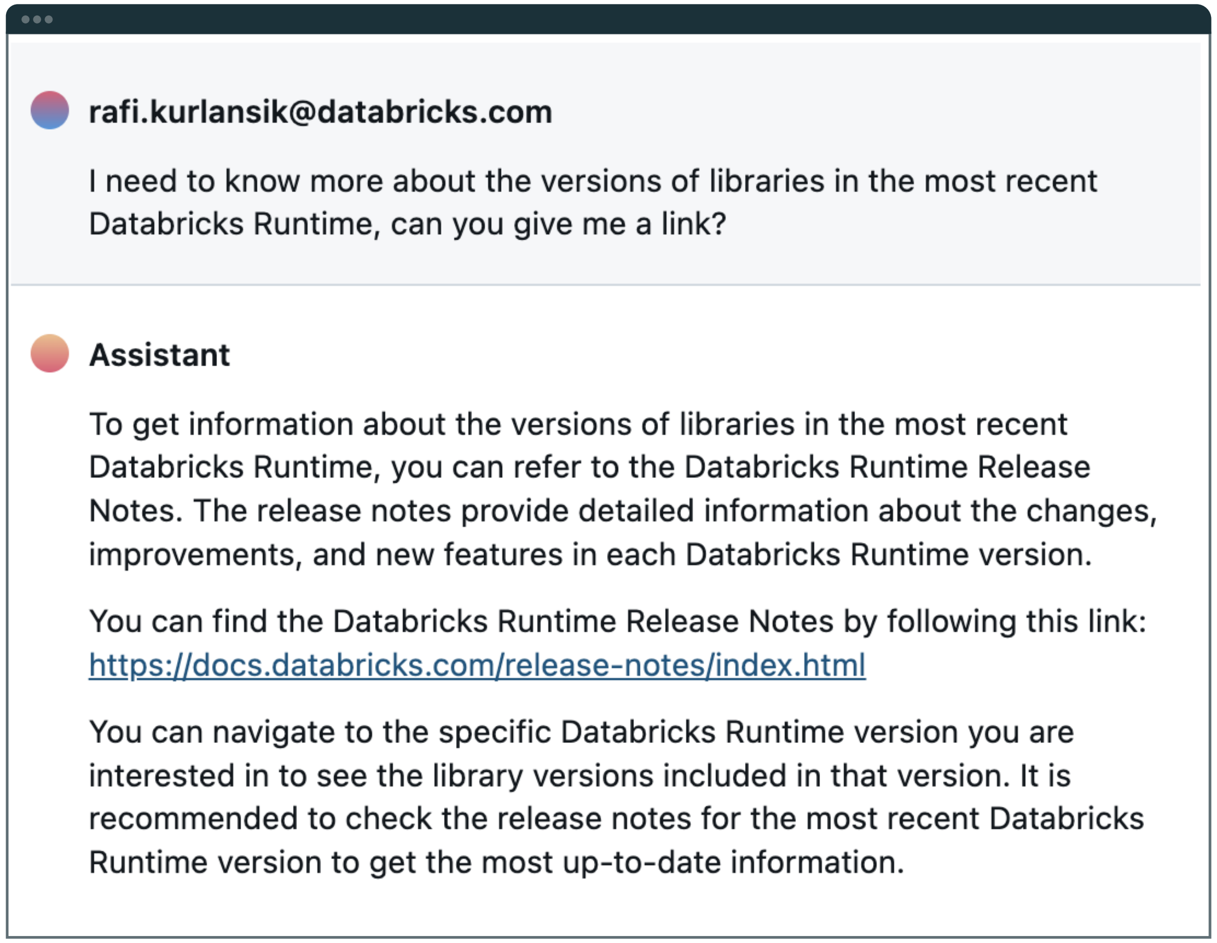
```
[Shift+Enter] to run and move to next cell
[Esc H] to see all keyboard shortcuts
```

This example highlights the fact that being specific and adding detail to your prompt yields better results. If we simply ask the Assistant to write tests for us without any detail, our results will exhibit more variability in quality. Being specific and clear in what we are looking for — a test using PySpark modules that builds off the logic it wrote — generally will perform better than assuming the Assistant can correctly guess at our intentions.

GETTING HELP

Beyond a general capability to improve and understand code, the Assistant possesses knowledge of the entire Databricks documentation and Knowledge Base. This information is indexed on a regular basis and made available as additional context for the Assistant via a RAG architecture. This allows users to search for product functionality and configurations without leaving the Databricks Platform.

For example, if you want to know details about the system environment for the version of Databricks Runtime you are using, the Assistant can direct you to the appropriate page in the Databricks documentation.



rafi.kurlansik@databricks.com

I need to know more about the versions of libraries in the most recent Databricks Runtime, can you give me a link?

Assistant

To get information about the versions of libraries in the most recent Databricks Runtime, you can refer to the Databricks Runtime Release Notes. The release notes provide detailed information about the changes, improvements, and new features in each Databricks Runtime version.

You can find the Databricks Runtime Release Notes by following this link: https://docs.databricks.com/release-notes/index.html

You can navigate to the specific Databricks Runtime version you are interested in to see the library versions included in that version. It is recommended to check the release notes for the most recent Databricks Runtime version to get the most up-to-date information.

The Assistant can handle simple, descriptive, and conversational questions, enhancing the user experience in navigating Databricks' features and resolving issues. It can even help guide users in filing support tickets! For more details, read the announcement article.

CONCLUSION

The barrier to entry for quality data engineering has been lowered thanks to the power of generative AI with the Databricks Assistant. Whether you are a newcomer looking for help on how to work with complex data structures or a seasoned veteran who wants regular expressions written for them, the Assistant will improve your quality of life. Its core competency of understanding, generating, and documenting code boosts productivity for data engineers of all skill levels. To learn more, see the Databricks documentation on how to get started with the Databricks Assistant today.

Applying Software Development and DevOps Best Practices to Delta Live Table Pipelines

by Alex Ott

Databricks Delta Live Tables (DLT) radically simplifies the development of the robust data processing pipelines by decreasing the amount of code that data engineers need to write and maintain. And also reduces the need for data maintenance and infrastructure operations, while enabling users to seamlessly promote code and pipelines configurations between environments. But people still need to perform testing of the code in the pipelines, and we often get questions on how people can do it efficiently.

In this chapter we'll cover the following items based on our experience working with multiple customers:

- How to apply DevOps best practices to Delta Live Tables
- How to structure the DLT pipeline's code to facilitate unit and integration testing
- How to perform unit testing of individual transformations of your DLT pipeline
- How to perform integration testing by executing the full DLT pipeline
- How to promote the DLT assets between stages
- How to put everything together to form a CI/CD pipeline (with Azure DevOps as an example)

APPLYING DEVOPS PRACTICES TO DLT: THE BIG PICTURE

The DevOps practices are aimed at shortening the software development life cycle (SDLC) providing the high quality at the same time. Typically they include these steps:

- Version control of the source code and infrastructure
- Code reviews
- Separation of environments (development/staging/production)
- Automated testing of individual software components and the whole product with the unit and integration tests
- Continuous integration (testing) and continuous deployment of changes (CI/CD)

All these practices can be applied to Delta Live Tables pipelines as well:

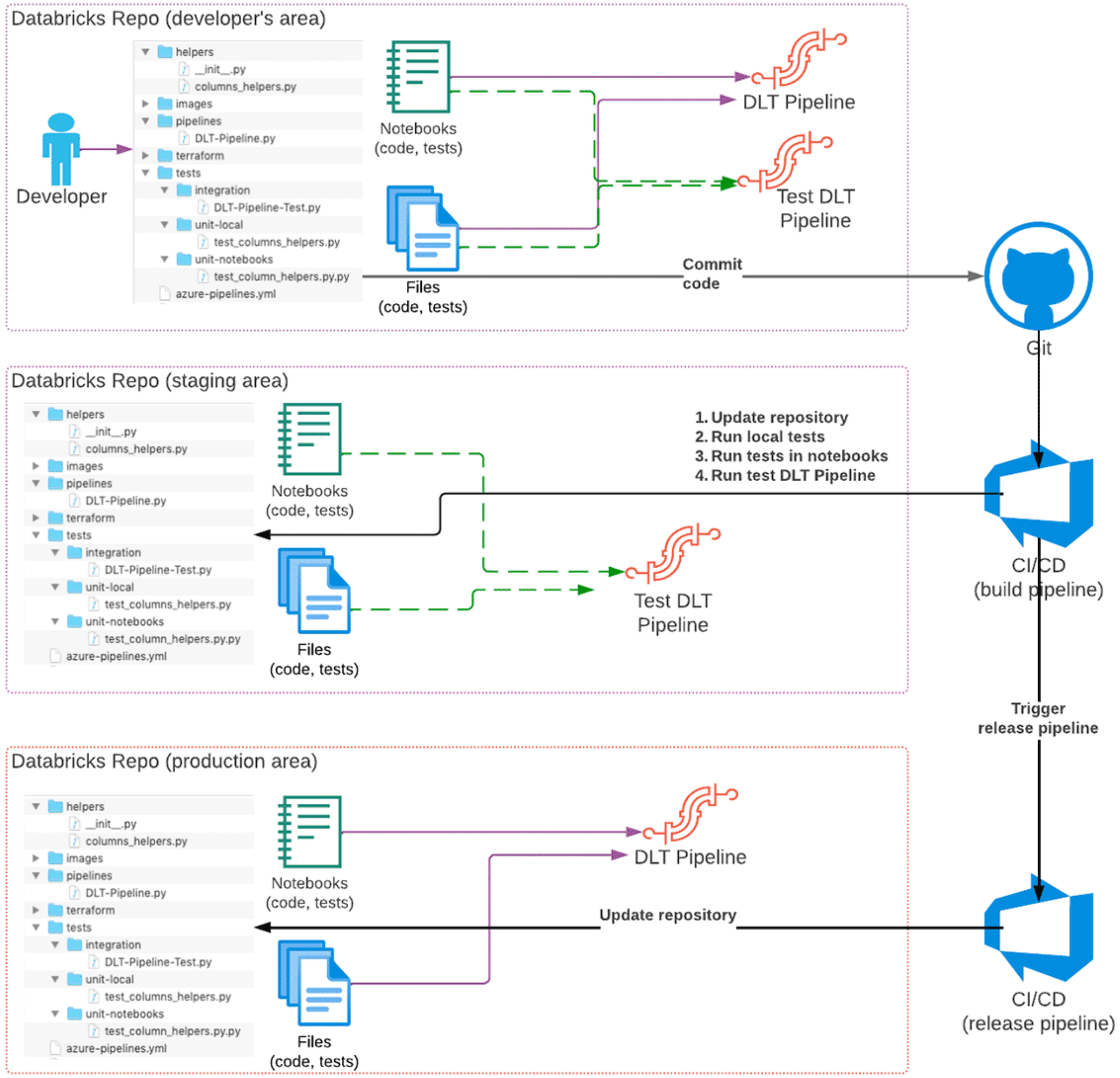
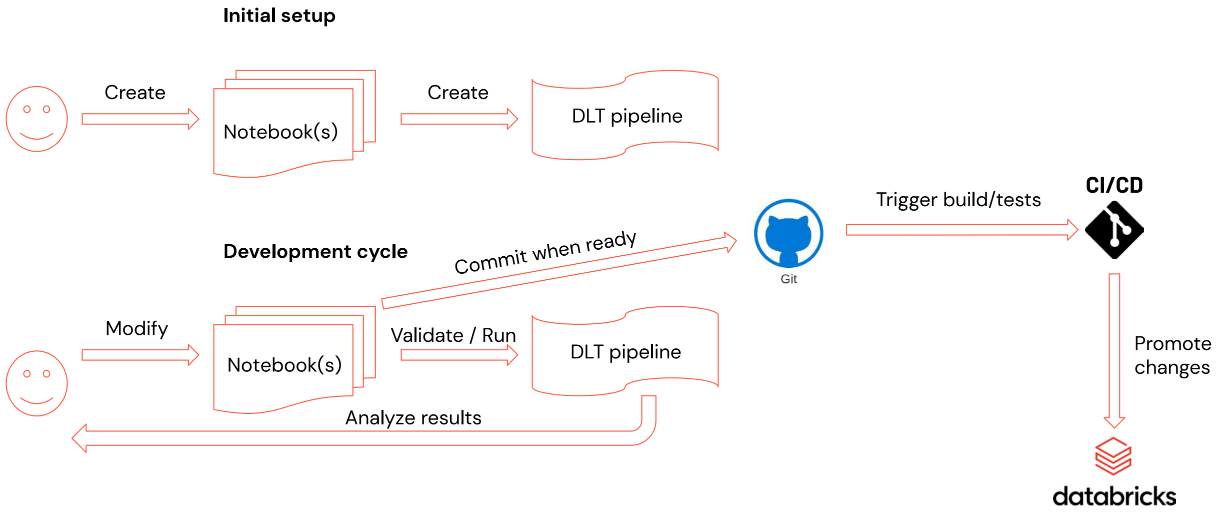


Figure: DLT development workflow

To achieve this we use the following features of Databricks product portfolio:

- Databricks Repos provide an interface to different Git services, so we can use them for code versioning, integration with CI/CD systems, and promotion of the code between environments
- Databricks CLI (or Databricks REST API) to implement CI/CD pipelines
- Databricks Terraform Provider for deployment of all necessary infrastructure and keeping it up to date

The recommended high-level development workflow of a DLT pipeline is as following:



1. A developer is developing the DLT code in their own checkout of a Git repository using a separate Git branch for changes.
2. When code is ready and tested, code is committed to Git and a pull request is created.
3. CI/CD system reacts to the commit and starts the build pipeline (CI part of CI/CD) that will update a staging Databricks Repo with the changes, and trigger execution of unit tests.
    a. Optionally, the integration tests could be executed as well, although in some cases this could be done only for some branches, or as a separate pipeline.
4. If all tests are successful and code is reviewed, the changes are merged into the main (or a dedicated branch) of the Git repository.
5. Merging of changes into a specific branch (for example, releases) may trigger a release pipeline (CD part of CI/CD) that will update the Databricks Repo in the production environment, so code changes will take effect when pipeline runs next time.

As illustration for the rest of the chapter we'll use a very simple DLT pipeline consisting just of two tables, illustrating typical Bronze/Silver layers of a typical lakehouse architecture. Complete source code together with deployment instructions is available on GitHub.

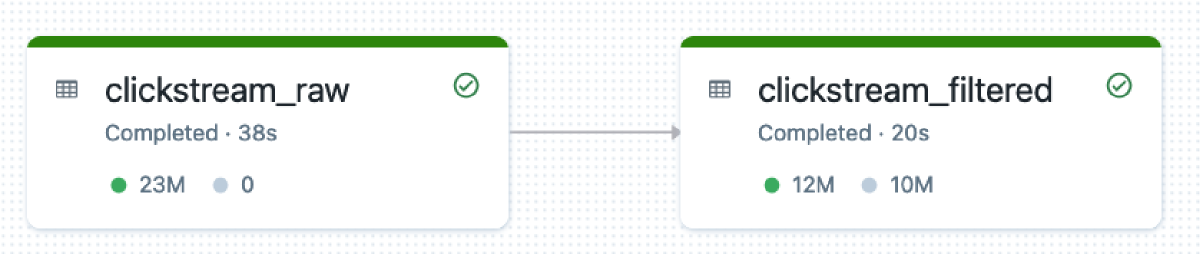


Figure: Example DLT pipeline

Note: DLT provides both SQL and Python APIs, in most of the chapter we focus on Python implementation, although we can apply most of the best practices also for SQL-based pipelines.

DEVELOPMENT CYCLE WITH DELTA LIVE TABLES

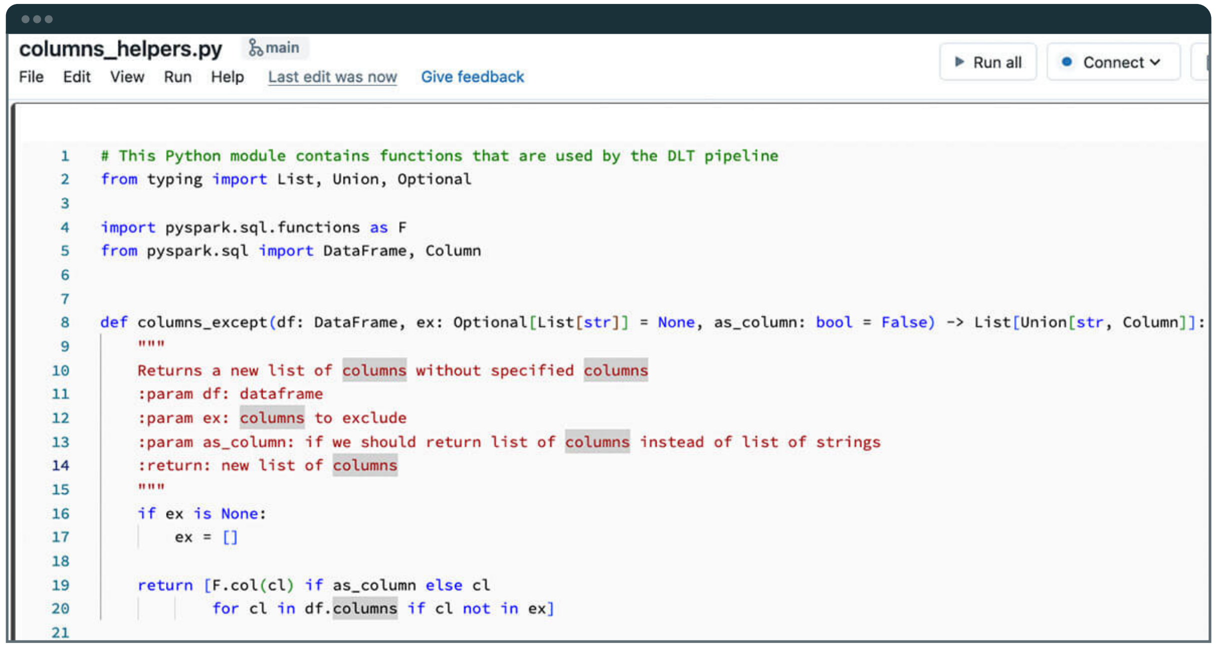
When developing with Delta Live Tables, typical development process looks as follows:

1. Code is written in the notebook(s).
2. When another piece of code is ready, a user switches to DLT UI and starts the pipeline. (To make this process faster it's recommended to run the pipeline in the Development mode, so you don't need to wait for resources again and again).
3. When a pipeline is finished or failed because of the errors, the user analyzes results, and adds/modifies the code, repeating the process.
4. When code is ready, it's committed.

For complex pipelines, such dev cycle could have a significant overhead because the pipeline's startup could be relatively long for complex pipelines with dozens of tables/views and when there are many libraries attached. For users it would be easier to get very fast feedback by evaluating the individual transformations and testing them with sample data on interactive clusters.

STRUCTURING THE DLT PIPELINE'S CODE

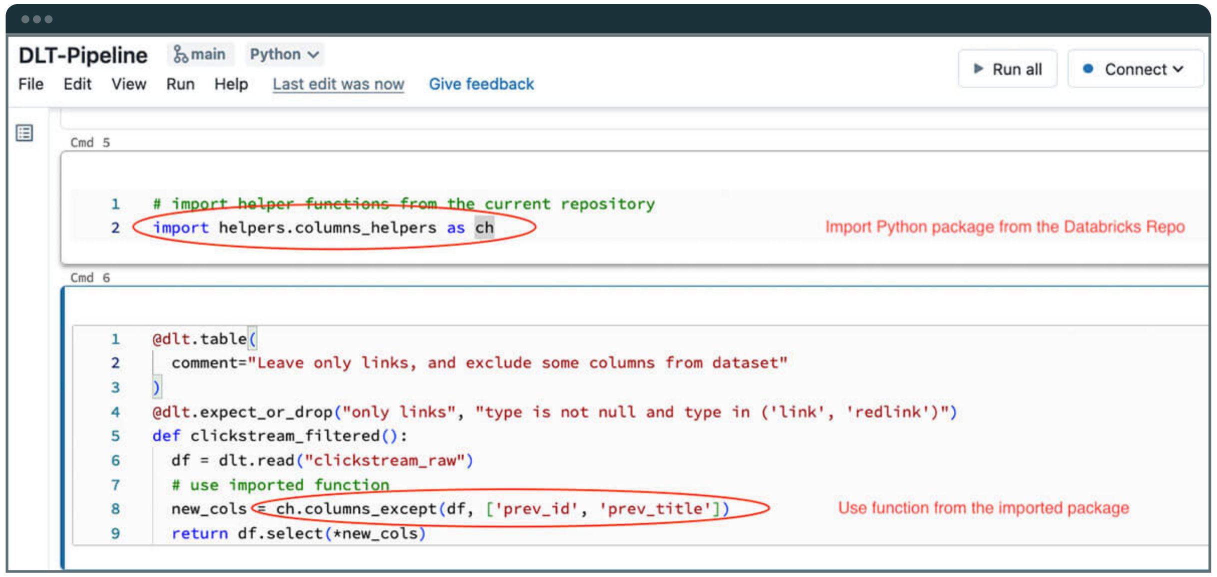
To be able to evaluate individual functions and make them testable it's very important to have correct code structure. Usual approach is to define all data transformations as individual functions receiving and returning Spark DataFrames, and call these functions from DLT pipeline functions that will form the DLT execution graph. The best way to achieve this is to use files in repos functionality that allows to expose Python files as normal Python modules that could be imported into Databricks notebooks or other Python code. DLT natively supports files in repos that allows importing Python files as Python modules (please note, that when using files in repos, the two entries are added to the Python's sys.path — one for repo root, and one for the current directory of the caller notebook). With this, we can start to write our code as a separate Python file located in the dedicated folder under the repo root that will be imported as a Python module:



```python
# This Python module contains functions that are used by the DLT pipeline
from typing import List, Union, Optional

import pyspark.sql.functions as F
from pyspark.sql import DataFrame, Column


def columns_except(df: DataFrame, ex: Optional[List[str]] = None, as_column: bool = False) -> List[Union[str, Column]]:
    """
    Returns a new list of columns without specified columns
    :param df: dataframe
    :param ex: columns to exclude
    :param as_column: if we should return list of columns instead of list of strings
    :return: new list of columns
    """
    if ex is None:
        ex = []

    return [F.col(cl) if as_column else cl
            for cl in df.columns if cl not in ex]

```

Figure: Source code for a Python package

And the code from this Python package could be used inside the DLT pipeline code:



```python
# import helper functions from the current repository
import helpers.columns_helpers as ch
```

Import Python package from the Databricks Repo

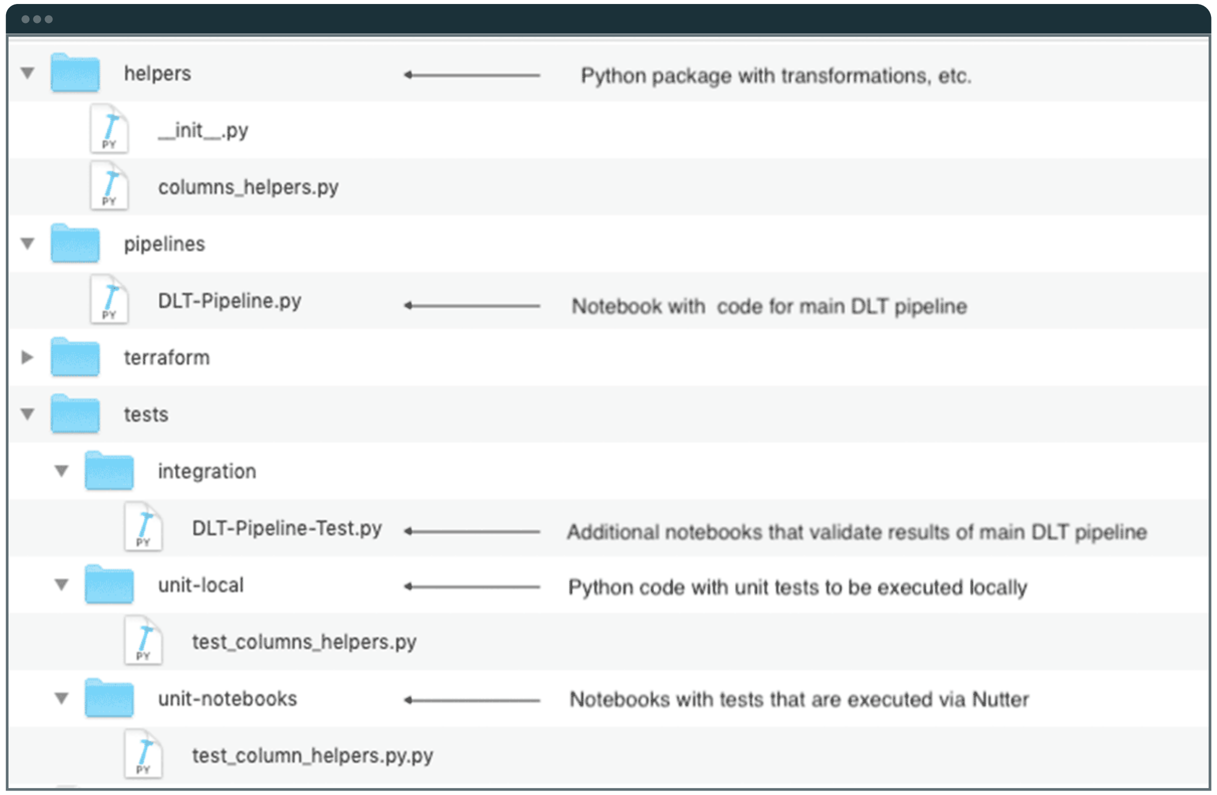
```python
@dlt.table(
  comment="Leave only links, and exclude some columns from dataset"
)
@dlt.expect_or_drop("only links", "type is not null and type in ('link', 'redlink')")
def clickstream_filtered():
  df = dlt.read("clickstream_raw")
  # use imported function
  new_cols = ch.columns_except(df, ['prev_id', 'prev_title'])
  return df.select(*new_cols)
```

Use function from the imported package

Figure: Using functions from the Python package in the DLT code

Note, that function in this particular DLT code snippet is very small — all it's doing is just reading data from the upstream table, and applying our transformation defined in the Python module. With this approach we can make DLT code simpler to understand and easier to test locally or using a separate notebook attached to an interactive cluster. Splitting the transformation logic into a separate Python module allows us to interactively test transformations from notebooks, write unit tests for these transformations and also test the whole pipeline (we'll talk about testing in the next sections).

The final layout of the Databricks Repo, with unit and integration tests, may look as following:



```
▼ helpers                          ← Python package with transformations, etc.
    __init__.py
    columns_helpers.py
▼ pipelines
    DLT-Pipeline.py                ← Notebook with  code for main DLT pipeline
▶ terraform
▼ tests
  ▼ integration
      DLT-Pipeline-Test.py         ← Additional notebooks that validate results of main DLT pipeline
  ▼ unit-local                     ← Python code with unit tests to be executed locally
      test_columns_helpers.py
  ▼ unit-notebooks                 ← Notebooks with tests that are executed via Nutter
      test_column_helpers.py.py
```

Figure: Recommended code layout in Databricks Repo

This code structure is especially important for bigger projects that may consist of the multiple DLT pipelines sharing the common transformations.