Efficient Data Processing: Constructing a Data Platform on Databricks

Ryan Chynoweth

7 min read · Jul 14, 2023

Notice—Please check out Databricks Assest Bundles for more information related to the topic here.

Efficiently processing data is top of mind for data leaders. Balancing data quality, costs, and processing speed is critical to ensuring the maximum return on investment for a given solution. Dillon Bostwick, a Senior Solution Architect at Databricks, discusses this trade-off in his blog, To Improve Data Availability, Think 'Right-Time' not 'Real-Time', that not all data assets need to be generated in "real-time", as it is more likely that you need to have the "right-time" mindset. However, maintaining the ability to process data as fast as possible is critical.

By utilizing Databricks' Delta Live Tables (DLT), engineers can effortlessly achieve the desired flexibility to adjust data pipelines' frequency for cost

optimization, without the need for code modifications. DLT leverages Spark Structured Streaming and Delta Lake, empowering the creation of reliable, maintainable, and testable data pipelines. This straightforward framework enables engineers to swiftly construct robust production pipelines. Enterprises can readily adopt DLT to facilitate the construction of both near real-time and batch data pipelines, however, it is crucial to plan for scaling development efforts. Ensuring adherence to best practices, proper data management, and avoiding redundant work are essential considerations in this process.

A centralized platform team plays a crucial role in establishing standardized practices, providing essential resources, and offering templates to enhance the time to market for engineering teams within an organization. In this blog, we will cover the following topics:

- Developing a Standard Platform Library
- Centralized Pipeline Services
- High-Level Roles and Responsibilities

By addressing these discussion points, organizations can establish robust foundations, streamline development processes, and ensure efficient data processing within their data platforms.

Developing a Platform and Library

The main objective of Delta Live Tables is to simplify the development and deployment of streaming, incremental, and batch pipelines. It achieves this by offering a declarative syntax through SQL and Python, making the process straightforward and intuitive. For example, users can write the following SQL code to create a live table:

```
CREATE OR REFRESH LIVE TABLE top_spark_referers
AS SELECT previous_page_title, click_count
FROM live.clickstream_prepared
WHERE current_page_title = 'Apache_Spark'
```

While DLT provides simplicity and user-friendliness, organizations often seek established methods and patterns to enforce best practices for specific tasks. Creating a custom Python library can be a solution to enforce coding best practices, streamline development time, and ensure adherence to set standards across all pipelines. As a practical example, connecting to a Kafka topic would require writing the following code each time without a standardized approach.

```
df = (spark.readStream
        .format("kafka")
        .option("kafka.sasl.jaas.config", f"kafkashaded.org.apache.kafka.common.
        .option("kafka.security.protocol", security_protocol)
        .option("kafka.sasl.mechanism", sasl_mechanism)
        .option("kafka.bootstrap.servers", self.kafka_bootstrap_servers)
        .option("subscribe", topic )
        .option("startingOffsets", starting_offsets )
        .option("failOnDataLoss", fail_on_data_loss)
        .load()
      )
```

The code above would need to be written in each pipeline requiring this task, which can be duplicative and more complex than necessary for many data personas. What if we could allow the users to write the code below instead?

```
from dlt_platform.connectors.kafka_connect import KafkaConnect

k = KafkaConnect(bootstrap_servers)
df = k.read_kafka_stream(spark, topic, username, security_protocol, sasl_mechani
```

The above code is possible by creating a custom Kafka Connect module library like the following:

```
from pyspark.sql.functions import *
from pyspark.sql.types import *

class KafkaConnect():

  def __init__(self, kafka_bootstrap_servers):
    """
    :param kafka_bootstrap_servers: The Kafka bootstrap servers to connect to
    """
    self.kafka_bootstrap_servers = kafka_bootstrap_servers


  def read_kafka_stream(self, spark, topic, username, security_protocol, sasl_me
    """
    Reads a given Kafka topic and returns a streaming dataframe.
    This function requires the exact options needed to read.

    Source: https://spark.apache.org/docs/2.1.1/structured-streaming-kafka-integ

    :param spark: Spark Object
    :param topic: the topic to read
    :param username: username to use for authentication
    :param security_protocol: encryption protocol
    :param sasl_mechanism: authentication mechanism
    :param starting_offsets: starting offset value. Default 'earliest'.
    :param fail_on_data_loss: Whether to fail the query when it's possible that
    :returns: Spark Streaming Dataframe representing the DLT table
    """
    return (spark.readStream
            .format("kafka")
            .option("kafka.sasl.jaas.config", f"kafkashaded.org.apache.kafka.com
            .option("kafka.security.protocol", security_protocol)
            .option("kafka.sasl.mechanism", sasl_mechanism)
```

```
            .option("kafka.bootstrap.servers", self.kafka_bootstrap_servers)
            .option("subscribe", topic )
            .option("startingOffsets", starting_offsets )
            .option("failOnDataLoss", fail_on_data_loss)
            .load()
           )


  def write_kafka_stream(self, df, key_col, value_cols, topic, checkpoint_locati
    """
    Writes a given Spark Streaming Dataframe to a Kafka topic

    :param df: Spark Streaming Dataframe to write to Kafka
    :param key_col: the column to represent the key value in Kafka
    :param value_cols: list of column names (e.g. ['col1', 'col2']) to send as t
    :param topic: name of topic to write to
    :param checkpoint_location: location of streaming checkpoint
    :param username: username to use for authentication
    :param security_protocol: encryption protocol
    :param sasl_mechanism: authentication mechanism
    """
    (df
     .select(col(key_col).alias("key"), to_json(struct([x for x in value_cols]))
     .writeStream
     .format("kafka")
     .option("kafka.sasl.jaas.config", f"kafkashaded.org.apache.kafka.common.sec
     .option("kafka.security.protocol", security_protocol)
     .option("kafka.sasl.mechanism", sasl_mechanism)
     .option("kafka.bootstrap.servers", self.kafka_bootstrap_servers)
     .option("checkpointLocation", checkpoint_location )
     .option("topic", topic)
     .start()
    )

  def generic_read_kafka_stream(self, spark, options):
      """
      Reads a given Kafka topic and returns a streaming dataframe.
      This function allows users to pass any options they wish.

      :param spark: Spark Object
      :param options: the options to read from the kafka topic
      :returns: Spark Streaming Dataframe representing the DLT table
      """
      assert type(options) == dict
      return (spark.readStream
              .format("kafka")
              .options(**options)
              .load()
             )
```

By abstracting common tasks, a platform team empowers other development teams to achieve the following benefits:

- **Improve Cycle Time**: Users no longer need to search through documentation to remember the precise syntax for each task. They can simply import the library and connect with just a few lines of code, accelerating development speed.
- **Ensure High Quality Code**: Abstracting common tasks ensures that development teams adhere to the organization's best practices. This helps maintain code quality and consistency across projects.
- **Streamline Updates Management**: Without a common library, updating specific tasks becomes challenging, requiring modifications in multiple instances. Leveraging a shared library allows engineers to make changes once, commit them, and roll them out to all pipelines simultaneously. This ensures that the code is always up to date and simplifies the management of updates.

By implementing these strategies, organizations can optimize development processes, enhance code quality, and streamline updates, leading to more efficient and robust pipelines. To see an example of how to develop a common library please refer to this repository. Please note that this library can be used for Delta Live Table pipelines and Spark Structured Streaming solutions.

Some of the key developer features being used are Python Imports in combination with Databricks Repos. In general, I would recommend

abstracting your code into a Python library then use Databricks Notebooks as the code entry point for your Databricks Jobs.

Organizational Structure

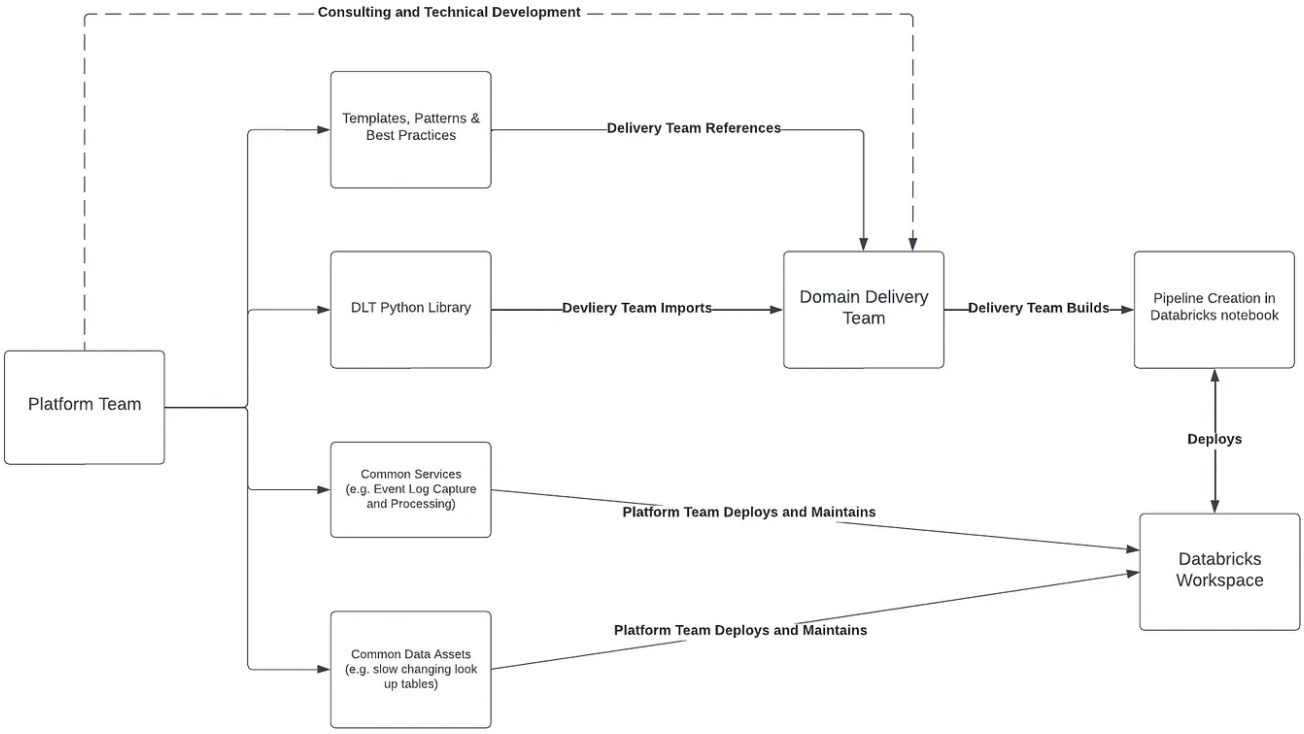
While decentralized ownership of data assets and solution delivery is prevalent in many organizations, excessive decentralization can result in redundant data, code, and effort. To ensure the enforcement and management of best practices, it is crucial to maintain a centralized team. Centralization facilitates the maintenance of global repositories that can be utilized across various domain delivery teams, promoting reusability and reducing duplication.

Developing a custom Delta Live Tables library empowers the platform team to:

- Minimize technical obstacles in constructing unified batch and streaming data pipelines.
- Enhance time to market for new solutions.
- Manage and sustain centralized infrastructure and reusable data assets.
- Implement centralized and automated monitoring and alerting systems for pipeline and solution health.
- Create templates, best practices, and comprehensive documentation for efficient development and maintenance.

The *platform team*'s role involves constructing reusable assets to expedite development while ensuring code quality and observability of deployed solutions. The *delivery team* collaborates with the business to collect requirements and translate them into technical specifications, then utilizing

the assets provided by the platform team they develop the solution. It is common for the platform team to tactically engage in technical consulting with the delivery teams to ensure requirements are met and to identify any new areas to extend the Platform library. Ultimately, the *business team* is responsible for defining solution requirements that drive maximum value for the organization.



Striking a balance between centralized management and decentralized ownership, organizations can harness the benefits of reusability, reduced duplication, improved time to market, and effective solution delivery. The collaboration between the platform team, delivery team, and business team ensures the alignment of technical requirements with business objectives, leading to the successful implementation of data pipelines and solutions that drive value for the organization.

Platform Shared Services

Shared Services owned by the platform team are essential for efficient and streamlined data pipeline development within any organization. Acting as a central hub, these services offer a range of reusable resources that promote consistency, scalability, and collaboration among teams. The platform team takes ownership of the following materials and services to enhance data pipeline development:

- **Custom Python Library**: A custom Python library reduces code duplication and promotes code reuse, ensuring high code quality across pipelines. It allows teams to leverage pre-built functions and modules, accelerating development while maintaining standards.
- **Deploying Common Data Services**: In a decentralized development environment, sharing data between processes becomes challenging. The centralized platform team can manage shared data services, providing a reliable and accessible data source for different consumer teams, promoting data consistency and reducing redundancy.
- **Shared Pipeline Monitoring**: Reusable monitoring solutions provided by the platform team enable delivery teams to ensure the health and performance of their data pipelines. Centralizing all Delta Live Table pipelines in the same storage account facilitates automated collection of new monitoring data, allowing teams to gain valuable insights and proactively address pipeline issues.
- **Templates and Reference Materials**: Providing templates and reference materials enables teams to kick-start their projects quickly and improve time to value. Access to comprehensive documentation and sample pipelines empowers teams to follow best practices, enhancing development speed and promoting standardized approaches.

These services contribute to the value provided by the platform team, positioning them as a center of excellence and a strategic hub for data-related initiatives within the organization. By leveraging these shared services, teams can enhance collaboration, maintain code quality, streamline development efforts, and expedite the delivery of data-driven solutions.

Conclusion

A centralized platform team serves as a catalyst for accelerating value generation and maximizing the return on investment for data solutions within an organization. To support your Databricks projects, you can refer to this repository, which provides an example of building a Python library. It's worth noting that similar approaches can also be applied to machine learning solutions. For instance, you can explore the forecasting example available here. By leveraging the expertise and resources of a centralized platform team, delivery and business teams can enhance their efficiency, expedite solution delivery, and drive greater value from their data initiatives.

Disclaimer: these are my own thoughts and opinions and not a reflection of my employer

Databricks | Data Streaming | Delta Live Tables | Spark Streaming | Python Libraries

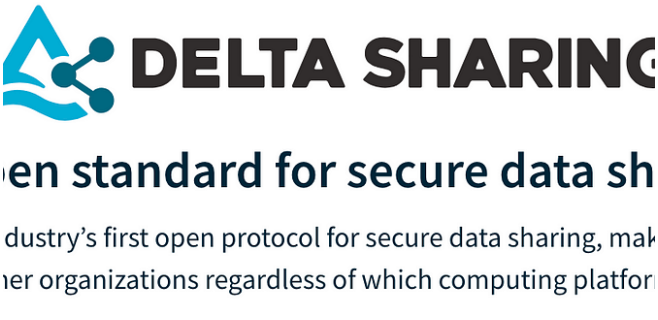
Written by Ryan Chynoweth

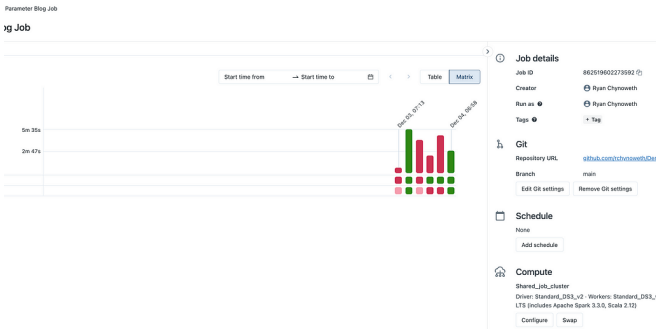
Edit profile

312 Followers

Senior Solutions Architect Databricks — anything shared is my own thoughts and opinions

More from Ryan Chynoweth





Ryan Chynoweth

Delta Sharing: An Implementation Guide for Multi-Cloud Architecture

Introduction

8 min read · 3 days ago

3 2

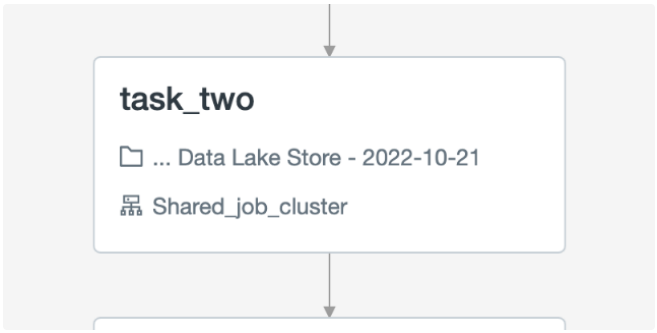
Ryan Chynoweth

Task Parameters and Values in Databricks Workflows

Databricks provides a set of powerful and dynamic orchestration capabilities that are...

11 min read · Dec 7, 2022

50 3



Ryan Chynoweth

Converting Stored Procedures to Databricks

Special thanks to co-author Kyle Hale, Sr. Specialist Solutions Architect at Databricks.

14 min read · Dec 29, 2022

116 5

Ryan Chynoweth

Recursive CTE on Databricks

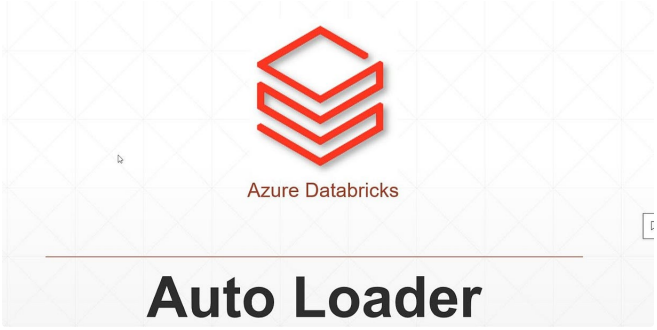
Introduction

3 min read · Apr 20, 2022

33

See all from Ryan Chynoweth

Recommended from Medium



Daan Rademaker

Do-it-yourself, building your own Databricks Docker Container

In my previous LinkedIn article, I aimed to persuade you of the numerous advantages o...

7 min read · Oct 16, 2023

26

SIRIGIRI HARI KRISHNA in Towards Dev

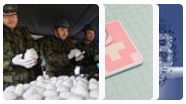
Auto Loader

Autoloader simplifies reading various data file types from popular cloud locations like...

5 min read · Dec 9, 2023

4 1

Lists

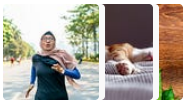


Staff Picks

547 stories · 597 saves

Stories to Help You Level-Up at Work

19 stories · 395 saves

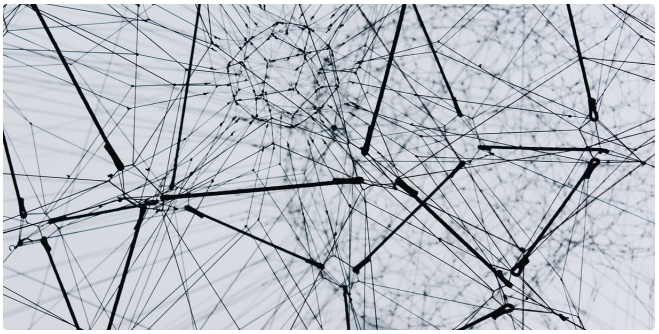


Self-Improvement 101

20 stories · 1146 saves

Productivity 101

20 stories · 1047 saves



Subham Khandelwal in Dev Genius

PySpark—Optimize Joins in Spark

Shuffle Hash Join, Sort Merge Join, Broadcast joins and Bucketing for better Join...

8 min read · 4 days ago

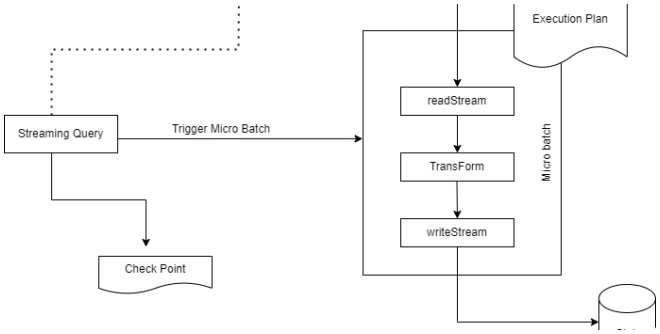
Manasreddy

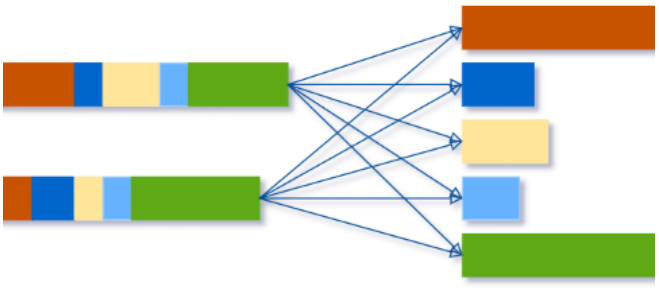
How to pass: Databricks Data Engineer Professional Certification

Conquering the Databricks Data Engineer Professional Exam: A Definitive Guide

3 min read · Aug 24, 2023

65 8





shorya sharma

Beyond Batch: Real-Time ETL with Spark Structured Streaming on...

In the dynamic landscape of big data analytics, the demand for real-time insights...

7 min read · Dec 25, 2023

Shantanu Tripathi

Why Shuffle is best served as External Service in Spark

Shuffle?

2 min read · Dec 21, 2023

6 52

See more recommendations