Highlighting Spline: Automated Data Lineage for Spark

Spline is an open-source tool designed specifically for automated data lineage tracking within Apache Spark jobs. It captures metadata about Spark transformations and provides a user interface for visualizing data flow. In your data platform, Spline provides critical visibility into how data is transformed from its raw sources through various Spark processing stages to its curated destinations, ensuring transparency and enabling easier debugging and governance. This guide will demonstrate basic and advanced use cases of Spline, leveraging your **Advanced Track** local environment setup and its integration with Spark and OpenMetadata.
Reference: This guide builds upon the concepts and setup described in **Section 4.2. Core Technology Deep Dive** of the **Core Handbook** and the **Progressive Path Setup Guide Deep-Dive Addendum**, specifically Spline's role in the **Orchestration & Governance Layer**.

Basic Use Case: Capturing and Visualizing Spark Job Lineage

Objective: To demonstrate how Spline automatically intercepts and records the execution plan of Spark jobs, making the lineage visible in its UI.
Role in Platform: Provide immediate, granular insights into Spark data transformations, helping developers understand data flow and verify changes.
Setup/Configuration (Local Environment - Advanced Track):

1. **Ensure all Advanced Track services are running:** docker compose up --build -d from your project root. This includes spark and spline.
2. **Verify Spline is accessible:** Navigate to http://localhost:8081 in your web browser.
3. **Ensure Spark is configured with Spline Agent:** Your docker-compose.yml for the spark service should include the Spline agent configuration.
   Example docker-compose.yml snippet for Spark + Spline:

```
# ...
spark:
  image: bitnami/spark:3.5.0
  # ... other config
  environment:
    # ... other Spark env vars
    # SPLINE CONFIGURATION
    SPARK_SUBMIT_ARGS: >
      --jars /opt/bitnami/spark/jars/spline-spark-agent-bundle_2.12-0.7.1.jar
      --driver-java-options
"-javaagent:/opt/bitnami/spark/jars/spline-agent-bundle-0.7.1.jar"
      --conf spark.spline.producer.url=http://spline:8081/producer
```

```
    --conf
spark.spline.persistence.factory=za.co.absa.spline.harvester.json.JsonLineagePersisten
ceFactory
    --conf spark.spline.mode=ENABLED
    --conf spark.spline.log.level=WARN # To reduce verbose logging
  volumes:
   - ./pyspark_jobs:/opt/bitnami/spark/jobs
   - spline_jars:/opt/bitnami/spark/jars # Mount spline jars if needed, or pre-built into
image
  depends_on:
   spline:
    condition: service_healthy
spline:
  image: absaspline/spline:0.7.1 # Use the correct version
  ports:
   - "8081:8081" # Spline UI
  environment:
   SPLINE_DATABASE_CONNECTION_URL: jdbc:h2:mem:spline;DB_CLOSE_DELAY=-1 #
Or use a persistent DB
   SPLINE_DATABASE_DRIVER: org.h2.Driver
   SPLINE_DATABASE_USER: sa
   SPLINE_DATABASE_PASSWORD: ""
  healthcheck:
   test: ["CMD", "curl", "-f", "http://localhost:8081/status || exit 1"]
   interval: 10s
   timeout: 5s
   retries: 5
# ...
volumes:
  spline_jars: # Define volume if you're externalizing jars
```

Note: The Spline agent JARs (spline-spark-agent-bundle_2.12-0.7.1.jar and spline-agent-bundle-0.7.1.jar) need to be present in the Spark container's classpath. The spline_jars volume is a conceptual way to manage this, or you might build a custom Spark image that includes them.

4. **Run a Spark Job:**
    - Submit one of your Spark jobs that writes to Delta Lake (e.g., pyspark_jobs/streaming_consumer.py or pyspark_jobs/batch_transformations.py).
    - For streaming_consumer.py:
      ```
      docker exec -it spark spark-submit \
        --packages
      org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.0,io.delta:delta-core_2.12:2.4.0 \
        --conf spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension \
      ```

```
  --conf
spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog
\
  --conf spark.hadoop.fs.s3a.endpoint=http://minio:9000 \
  --conf spark.hadoop.fs.s3a.access.key=minioadmin \
  --conf spark.hadoop.fs.s3a.secret.key=minioadmin \
  --conf spark.hadoop.fs.s3a.path.style.access=true \
  /opt/bitnami/spark/jobs/streaming_consumer.py \
  raw_financial_transactions kafka:29092
s3a://raw-data-bucket/financial_data_delta
```

- Let it run for a few seconds/minutes to generate lineage.

Steps to Exercise:

1. **Access Spline UI:** Open your web browser and go to http://localhost:8081.
2. **View Executions:**
   - On the main Spline UI page, you should see a list of recent Spark job "Executions."
   - Find the entry corresponding to the Spark job you just submitted (e.g., KafkaToDeltaStream_raw_financial_transactions or BatchETLTransformation).
   - Click on the execution.
3. **Explore Lineage Graph:**
   - Spline will display a visual graph representing the data lineage for that Spark job.
   - **Observe:** The graph typically shows:
     - **Sources:** Input data assets (e.g., Kafka topic raw_financial_transactions).
     - **Transformation:** A node representing the Spark job itself (the process).
     - **Destinations:** Output data assets (e.g., Delta Lake table raw-data-bucket/financial_data_delta).
   - You can click on nodes and edges to see more details about the data, schema, and transformation operations.

Verification:

- **Spline UI:** A clear and accurate visual lineage graph is displayed for the executed Spark job, showing the source(s), the Spark transformation process, and the destination(s). This confirms Spline is correctly capturing and visualizing Spark lineage.

Advanced Use Case 1: Detailed Schema and Operation Tracking

Objective: To demonstrate how Spline captures not just the overall data flow, but also detailed information about schema evolution and the specific Spark operations performed within a job.

Role in Platform: Provide forensic detail for debugging schema drift issues, understanding the exact transformations applied to data, and validating data contract adherence.

Setup/Configuration:

1. **Ensure Basic Use Case setup is complete.**

2. **Run a Spark Job with Transformations:** Submit a job that involves multiple steps or schema changes, such as pyspark_jobs/batch_transformations.py, which reads raw Delta, performs joins/transformations, and writes to curated Delta.
   ```
   docker exec -it spark spark-submit \
     --packages io.delta:delta-core_2.12:2.4.0,org.postgresql:postgresql:42.6.0 \
     --conf spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension \
     --conf
   spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog \
     --conf spark.hadoop.fs.s3a.endpoint=http://minio:9000 \
     --conf spark.hadoop.fs.s3a.access.key=minioadmin \
     --conf spark.hadoop.fs.s3a.secret.key=minioadmin \
     --conf spark.hadoop.fs.s3a.path.style.access=true \
     /opt/bitnami/spark/jobs/batch_transformations.py \
     s3a://raw-data-bucket/financial_data_delta \
     s3a://curated-data-bucket/financial_data_curated_batch
   ```

   (Ensure raw-data-bucket/financial_data_delta has some data).

Steps to Exercise:

1. **Access Spline UI:** http://localhost:8081.
2. **Select the Batch Transformation Execution:** Find the execution for BatchETLTransformation.
3. **Explore "Schema" and "Operations" Details:**
   - **Schema Tab:** Click on the input (e.g., s3a://raw-data-bucket/financial_data_delta) and output (e.g., s3a://curated-data-bucket/financial_data_curated_batch) nodes in the lineage graph. You should see a detailed view of the schema (column names, types) for each dataset. Compare how the schema changes from raw to curated.
   - **Operations Tab:** Click on the central Spark job (process) node. This tab provides a breakdown of the logical plan of the Spark job. You'll see individual operations like LogicalRelation, Project, Filter, Join, Aggregate, SaveIntoDataSourceCommand.
   - **Drill Down:** Click on individual operations to see their specific parameters and the attributes (columns) they affect. For example, a Join operation will show the join condition, and a Project operation will show the selected and transformed columns.

Verification:

- **Spline UI:** The "Schema" tab accurately displays the schema for input and output datasets, reflecting transformations. The "Operations" tab details the logical plan of the Spark job, including specific transformations like joins and projections, demonstrating Spline's deep insight into Spark's execution.

Advanced Use Case 2: Integrating Lineage with

OpenMetadata

Objective: To demonstrate how Spline's collected lineage data is pushed to OpenMetadata, providing a unified view of data assets and their end-to-end data flow within the central data catalog.
Role in Platform: Centralize lineage information from various processing engines (starting with Spark), enabling a holistic understanding of data dependencies for governance, impact analysis, and compliance.
Setup/Configuration:

1. **Ensure Basic Use Case and Advanced Use Case 1 (Spline data collection) are working.**
2. **Ensure OpenMetadata is running:** http://localhost:8585.
3. **Ensure OpenMetadata Spline Connector is configured and running:** Your Airflow DAGs (e.g., openmetadata_ingestion_dag) should include a task to run the OpenMetadata Spline connector, which pulls lineage from Spline and ingests it into OpenMetadata.
    - This typically involves running a script in openmetadata_ingestion_scripts/ which uses the OpenMetadata Python client.
    - You might need to manually trigger this Airflow DAG if it's not on a schedule.

Steps to Exercise:

1. **Run a Spark Job:** Submit any Spark job that writes to Delta Lake (e.g., streaming_consumer.py) to ensure fresh lineage is generated by Spline.
2. **Trigger OpenMetadata Lineage Ingestion:** Manually trigger the relevant Airflow DAG in the Airflow UI (http://localhost:8080) that runs the OpenMetadata Spline connector.
3. **Access OpenMetadata UI:** Go to http://localhost:8585.
4. **Search for an Output Table:** Search for the Delta Lake table that was the destination of your Spark job (e.g., raw-data-bucket.financial_data_delta).
5. **Navigate to the "Lineage" Tab:**
    - On the table's detail page, click the "Lineage" tab.
    - **Observe:** You should see a graphical representation of the lineage, showing the Kafka topic as a source, the Spark job as a process, and the Delta table as the destination. This lineage is pulled from Spline via the OpenMetadata connector.
    - **Explore:** Hover over nodes to see details, and if configured, you might see column-level lineage within OpenMetadata, showing how source columns map to target columns.

Verification:

- **OpenMetadata UI:** The "Lineage" tab for the Spark-generated Delta Lake table correctly displays the lineage graph, confirming that OpenMetadata successfully ingested the lineage metadata from Spline. This is a crucial integration point for comprehensive data governance.

Advanced Use Case 3: Customizing Lineage and

Event-Driven Lineage Capture (Conceptual)

Objective: To conceptually discuss how Spline can be customized (e.g., by adding custom attributes to lineage events) and how lineage capture can be made more event-driven, rather than relying solely on post-execution polling.
Role in Platform: Extend lineage capabilities to include custom business metadata, and enable more real-time updates to the data catalog's lineage graph.
Setup/Configuration (Conceptual Discussion):

1. **Custom Attributes:**
    - Spline allows adding custom "extra" metadata to lineage events. This can be done by configuring the Spline agent to read specific Spark properties or by modifying the Spark job itself to set these properties.
    - Example: Add a spark.spline.extra.tags=batch_id:123,data_owner:data_team
2. **Event-Driven Lineage (beyond current stable Spline):**
    - While the current Spline version typically relies on the Spark agent sending lineage after a job completes, future developments or custom integrations could involve more real-time eventing.
    - Conceptually, a lightweight service could listen for Spark "job completion" events (e.g., from an Airflow XCom or Spark listener API) and immediately trigger a push of that job's lineage from Spline to OpenMetadata, reducing latency in metadata updates.

Steps to Exercise (Conceptual/Discussion):

1. **Discuss Customizing Lineage:**
    - Explain how you might configure Spark jobs to emit custom attributes (e.g., project_name, pipeline_id, business_domain) as part of the Spline lineage.
    - Show how these custom attributes would then appear in the Spline UI when you inspect an execution's details, providing richer context for data governance.
    - Discuss how this allows data teams to tailor lineage to their specific organizational needs and reporting requirements.
2. **Discuss Event-Driven Lineage Capture:**
    - Explain the benefits of real-time lineage updates (e.g., faster visibility in OpenMetadata, more responsive impact analysis).
    - Describe a conceptual architecture where:
        - A Spark custom listener (or a mechanism in Airflow) detects Spark job completion.
        - An event (e.g., a message to Kafka/SQS) is sent with the Spline execution ID.
        - A lightweight Lambda or a small Airflow task consumes this event and immediately triggers the OpenMetadata Spline connector for that specific execution ID, rather than waiting for a scheduled poll.

Verification (Conceptual):

- **Enhanced Context:** The ability to add custom attributes to lineage demonstrates how the data platform can capture business-specific context, making lineage more

meaningful for various stakeholders.

- **Real-time Potential:** Understanding event-driven lineage patterns highlights the path toward more immediate and dynamic data governance capabilities, aligning with real-time operational needs.

This concludes the guide for Spline.