SPIP: Declarative Pipeline Framework for Apache Spark

Apr 5, 2025

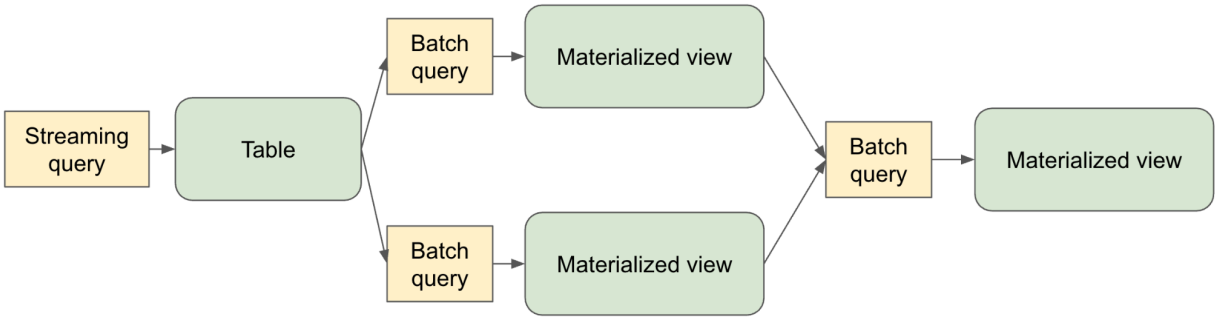
Sandy Ryza, Chao Sun, Yuming Wang, Kent Yao, Jie Yang
Shepherd: Hyukjin Kwon

Q1. What are you trying to do? Articulate your objectives using absolutely no jargon.

We propose a new abstraction that combines multiple Apache Spark transformations into a single declarative dataflow graph, to simplify the development and management of data pipelines. This approach extends Spark's lazy, declarative execution model beyond single queries, to pipelines that keep multiple datasets up to date. It supports a shift in development from imperative coding ('how' to execute) to declarative specification ('what' needs to exist), reducing cognitive overhead and manual orchestration of dependencies.

One of Apache Spark's most common use cases is building data pipelines: sets of connected data flows that derive downstream datasets from upstream source data. Today, developing data pipelines is difficult and error-prone. Pipeline developers share a common set of challenges around ordering execution of data flows, catching boundary issues, and executing pipelines efficiently.

Fig: example pipeline



The Declarative Pipelines framework for Apache Spark will address these challenges by enabling users to build data pipelines by using familiar Spark SQL & DataFrame APIs within a declarative framework. Inspired by frameworks like dbt, SQLMesh, Flink Tables, and DLT, it will include APIs that allow defining a full graph of tables and data flows prior to launching any

execution. Understanding the dataflow graph ahead of execution enables efficient parallel execution and catching bugs early.

We will provide the framework across Spark's supported languages, including Python, SQL, and Scala/Java. In addition to the public API, we will provide a pipeline runner for parallel dataflow graph execution.

Q2. What problem is this proposal NOT designed to solve?

Declarative Pipelines is not intended to replace general-purpose orchestration tools like Airflow, Dagster, or Prefect: it won't include general-purpose orchestration features like scheduling or history-tracking. Rather, Declarative Pipelines will be able to be embedded as tasks inside general-purpose orchestration DAGs.

Q3. How is it done today, and what are the limits of current practice?

Today, users ingest data from disparate sources and perform a series of transformations between intermediary storage layers before sending the results to their final destinations. Those sources and destinations include message buses, cloud storage, data lakes, data warehouses, databases, monitoring tools, and more.

These multi-step pipelines introduce complexity that has to be managed outside of each individual flow. Common challenges include:

- Updating tables in the right order – if a step in a data pipeline reads data from a table before another step has a chance to update it, it's easy for downstream data to end up stale and out-of-sync in unexpected ways. Manually managing execution dependencies is tedious and error-prone.
- Handling parallelism, while respecting data dependencies, across multiple queries. Steps in a pipeline are often executed sequentially, even when they don't have data dependencies, because it's easier to implement. This results in higher end-to-end latency and lower resource utilization.
- Quickly identifying errors at the boundaries between data processing steps, such as a downstream transformation expecting columns that don't exist in an upstream table.

Q4. What is new in your approach and why do you think it will be successful?

Q4a. What is new in your approach?

Developing data pipelines

The Declarative Pipelines API will enable users to develop pipelines that describe their data processing graph end-to-end.

The proposed APIs are based on Databricks DLT APIs, which have been battle-tested through several iterations of user feedback and revision. While DLT is tied to the Delta table format and a restricted set of data catalogs, the Spark Declarative Pipelines proposal generalizes to all the data formats and catalogs that Spark supports. Some strategies for incremental computation will depend on features that are only available in table formats, like Iceberg, Delta, and Hudi. These features will be accessed through Spark's data source APIs, and will not favor or be tied to any particular format.

A declarative pipeline is composed of declarations of flows, tables, materialized views, and external sinks.

- A **flow** is a data processing step: each flow is responsible for updating the contents of a particular object in persistent storage, like a table. Except in the cases where multiple flows target a single object, flows are typically defined in the same statement that defines the object they target.
- A **materialized view** is an object containing the results of a query in physical storage. From the perspective of the Pipelines framework, a materialized view is a combination of a storage location and a "batch" flow that, when executed, updates the data inside the storage location to contain the results of the query.
- A **streaming table** is a combination of a table and a flow that appends to that table by processing source data one record at a time.
- An **external sink** is a generic target for a flow to send data that is external to the pipeline. Because sinks are external to the pipeline, we can't always provide the idempotency semantics that we can with streaming tables and materialized views.

Below is an example of a pipeline that uses the Pipelines Python API to transform raw clickstream data into top page referrers. It defines a streaming table that holds the clickstream data, as well as several materialized views built on top of it.

Python

```
from pyspark import pipelines
```

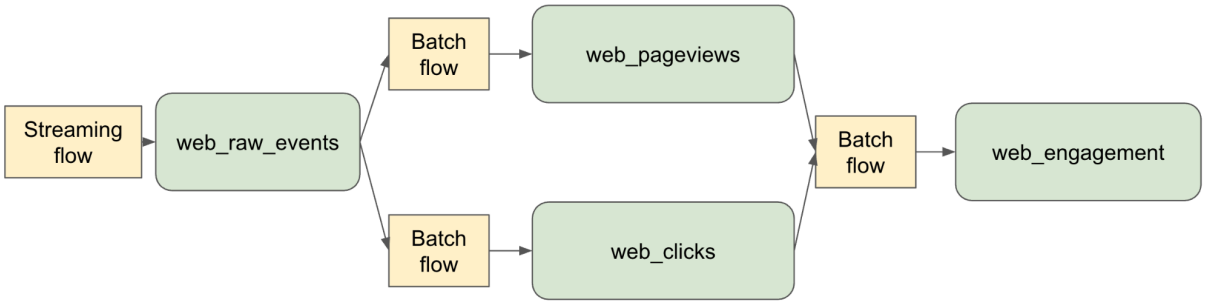
```python
# Defines a streaming table, i.e. a table targeted by a flow that reads data from
a
# stream
@pipelines.table(comment="The raw wikipedia clickstream dataset.")
def clickstream():
    return spark.readStream.format("json").load("clickstream.json")

# Defines a materialized view that reads data from the clickstream table above
@pipelines.materialized_view()
def top_spark_internal_referrers():
    return spark.read.table("clickstream") \
      .filter(expr("current_page_title == 'Apache_Spark'")) \
      .filter(expr("type == 'Link'")) \
      .sort(desc("click_count")) \
      .select("referrer", "click_count") \
      .limit(10)

# Defines a materialized view that reads data from the clickstream table above.
# Because the query that defines this materialized view doesn't depend on the
# top_spark_internal_referrers materialized view, the framework can execute their
# queries in parallel.
@pipelines.materialized_view()
def top_spark_external_referrers():
    return spark.read.table("clickstream") \
      .filter(expr("current_page_title == 'Apache_Spark'")) \
      .filter(expr("type == 'External'")) \
      .sort(desc("click_count")) \
      .select("referrer", "click_count") \
      .limit(10)

# Defines a materialized view that reads data from both of the materialized views
above.
@pipelines.materialized_view(
  name="main_catalog.gold.top_spark_referrers"
  comment="A materialized view containing the top pages linking to the Apache
Spark page."
)
def top_spark_referrers():
    top_internal = spark.read.table("top_spark_internal_referrers") \
        .withColumn("type", lit("Internal"))
    top_external = spark.read.table("top_spark_external_referrers") \
        .withColumn("type", lit("External"))
    return top_internal.union(top_external)
```

This pipeline is represented in the following diagram, in which tables and materialized views are colored green and flows are colored yellow:



Below is an example that uses SQL to define the same pipeline:

```
None

CREATE OR REPLACE STREAMING TABLE clickstream AS SELECT * FROM
json.`clickstream.json`;


CREATE OR REPLACE MATERIALIZED VIEW top_spark_internal_referrers AS
SELECT referrer, click_count
FROM clickstream
WHERE current_page_title = 'Apache_Spark' AND type = 'Link'
ORDER BY click_count DESC
LIMIT 10;


CREATE OR REPLACE MATERIALIZED VIEW top_spark_external_referrers AS
SELECT referrer, click_count
FROM clickstream
WHERE current_page_title = 'Apache_Spark' AND type = 'External'
ORDER BY click_count DESC
LIMIT 10;


CREATE OR REPLACE MATERIALIZED VIEW main_catalog.gold.top_spark_referrers AS
SELECT referrer, click_count, 'Internal' AS type FROM top_spark_internal_referrers
UNION ALL
SELECT referrer, click_count, 'External' AS type FROM
top_spark_external_referrers;
```

Below is an example pipeline that uses an append flow and sink, but no internal datasets:

```python
from pyspark import pipelines

# example sink representing an Apache Kafka topic
pipelines.create_sink(
    format="kafka",
    name="myKafkaTopic",
    topic="topic1",
    options={
        "kafka.bootstrap.servers": "host1:port1,host2:port2",
        "kafka.security.protocol": "SASL_SSL",
    }
)

# example flow targeting the kafka sink
@pipelines.append_flow(
    target="myKafkaTopic",
    name="flow1", # optional, defaults to function name
)
def appendKafkaFlow():
  return spark.readStream.format("delta").table("catalog.schema.table")
```

Executing data pipelines

There can be two modes for executing pipelines:

- Triggered mode: all streaming flows are executed to process the data that's available at the start of the update, and all materialized views are updated to reflect the latest results of their query.
- Continuous mode: streaming flows are run to continuously process incoming data. Materialized views are updated at regular intervals when their inputs change.

We will add a new `spark-pipelines` CLI, built on top of `spark-submit`, which assembles and executes a pipeline. An example usage:

```
./bin/spark-pipelines run \
  --pipeline-conf my_pipeline.yaml
```

As with spark-submit, these arguments will be able to be included in config files so they don't need to be included on the command line every time. The command references a configuration file that defines the set of source files that declare the flows that compose the pipeline:

```
None
name: my_pipeline
schema: my_schema
catalog: my_catalog
configuration:
  spark.sql.shuffle.partitions: "1000"
libraries:
  - file:
      path: /my/path/clean_up.py
  - file:
      path: /my/path/pre-aggregate.sql
```

Q4b. Why do you think it will be successful?

The Declarative Pipelines API and framework help address the challenges mentioned in question two in the following ways:

Challenge	**How declarative pipelines helps**
Updating tables in the right order	Automatically analyzes data dependencies to determine execution order
Handling parallelism	Automatically executes updates in parallel when they don't have data dependencies
Quickly identifying errors at the boundaries between data processing steps	Automatically analyzes the entire graph before execution to find cycles, schema incompatibilities, and other inconsistencies. E.g., in the example above, if the query that defines the top_spark_internal_referrers materialized view depends on a column that's not produced by the query that creates the clickstream table, the framework can surface this before executing either query.

Q5. Who cares? If you are successful, what difference will it make?

One of Spark's most common use cases is building data pipelines. Providing a first-party declarative pipelines API will dramatically simplify this use case for Spark developers. It will also make building data pipelines with Spark accessible to a wider audience: e.g. developers who

are primarily familiar with writing data transformations using SQL will be able to use Spark to write performant data pipelines.

Q6. What are the risks?

Maximizing pipeline flow execution efficiency requires taking advantage of incremental data processing, which relies on an understanding of the changes in the underlying tables. The current Data Source v2 API only has limited support for this, so expanding the Data Source V2 API to expose more of the capabilities of table formats like Iceberg/Delta/Hudi will be required to take full advantage of the pipeline model.

Q7. How long will it take?

The expected timeframe for implementing the new APIs and a basic pipeline runner is three months. The full functionality described at a high-level in this document will take six to twelve months.

Q8. What are the mid-term and final “exams” to check for success?

- Mid-term: new Pipelines APIs are all implemented, but function as no-ops.
- Final: pipeline runner that can execute the flows in the pipeline in parallel, respecting data dependencies.