OTHER FACTORS TO CONSIDER

There are other factors you may need to plan for in your deployment. These are outside the scope of this post, but you will need to consider them as part of implementation. Among these are:

1. Authentication requirements of the target API: It is likely that the REST API will require authentication. This is typically done by adding required headers in your code before making the HTTP POST.
2. Potential rate limiting: The target REST API may implement rate limiting which will place a cap on the number of calls you can make to it per second or minute. You will need to ensure you can meet throughout targets within this limit. You'll also want to be ready to handle throttling errors that may occur if the limit is exceeded.
3. Network path required from worker subnet to target API: Obviously, the worker nodes in the host Spark cluster will need to make HTTP calls to the REST API endpoint. You'll need to use the available cloud networking options to configure your environment appropriately.
4. If you control the implementation of the target REST API (e.g., an internal custom service), be sure the design of that service is ready for the load and throughput generated by the streaming workload.

MEASURED THROUGHPUT TO A MOCKED API WITH DIFFERENT NUMBERS OF PARALLEL TASKS

To provide representative data of scaling REST API calls as described here, we ran tests using code very similar to Example 1 against a mocked up REST API that persisted data in a log.

Results from the test are shown in Table 1. These metrics confirm near-linear scaling as the task count was increased (by changing the partition count using repartition). All tests were run on the same cluster with a single 16-core worker node.

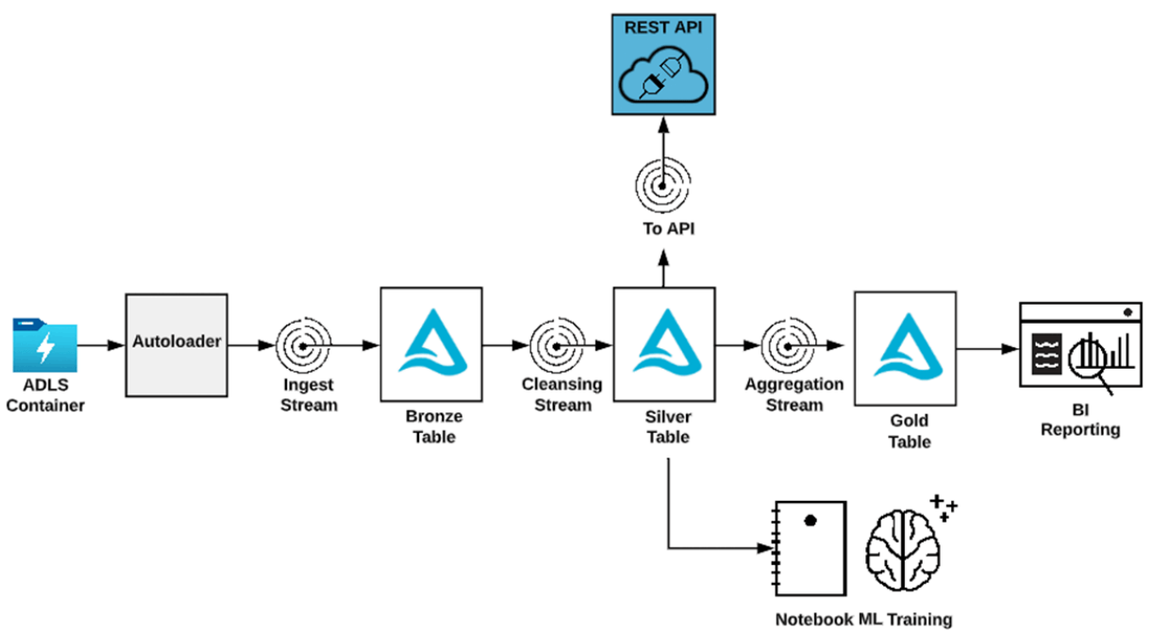
Rows per API Call	Partitions (threads)	Total calls	Seconds to complete processing	Rows per second	Calls per second	Calls per thread per second	Apprx API Latency (ms)
1	4	21,932	179	122	122	30.5	34
	8	21,932	88	249	249	31.1	33
	16	21,932	49	447	447	27.9	38
10	4	21,932	48	456	45	11.3	91
	8	21,932	25	877	87	10.9	100
	16	21,932	13	1687	168	10.5	100

Table 1

REPRESENTATIVE ALL UP PIPELINE DESIGNS

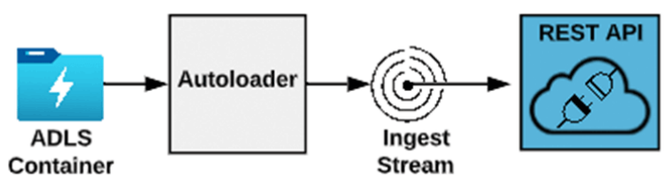
1. Routing some records in a streaming pipeline to REST API (in addition to persistent sinks)

   This pattern applies in scenarios where a Spark-based data pipeline already exists for serving analytics or ML use cases. If a requirement emerges to post cleansed or aggregated data to a REST API with low latency, the technique described here can be used.



2. Simple Autoloader to REST API job

   This pattern is an example of leveraging the diverse range of sources supported by Structured Streaming. Databricks makes it simple to consume incoming near real-time data - for example using Autoloader to ingest files arriving in cloud storage. Where Databricks is already used for other use cases, this is an easy way to route new streaming sources to a REST API.



SUMMARY

We have shown here how structured streaming can be used to send streamed data to an arbitrary endpoint - in this case, via HTTP POST to a REST API. This opens up many possibilities for flexible integration with analytics data pipelines. However, this is really just one illustration of the power of foreachBatch in Spark Structured Streaming.

The foreachBatch sink provides the ability to address many endpoint types that are not among the native sinks. Besides REST APIs, these can include databases via JDBC, almost any supported Spark connector, or other cloud services that are addressable via a helper library or API. One example of the latter is pushing data to certain AWS services using the boto3 library.

This flexibility and scalability enables Structured Streaming to underpin a vast range of real-time solutions across industries.

If you are a Databricks customer, simply follow the getting started tutorial to familiarize yourself with Structured Streaming. If you are not an existing Databricks customer, sign up for a free trial.

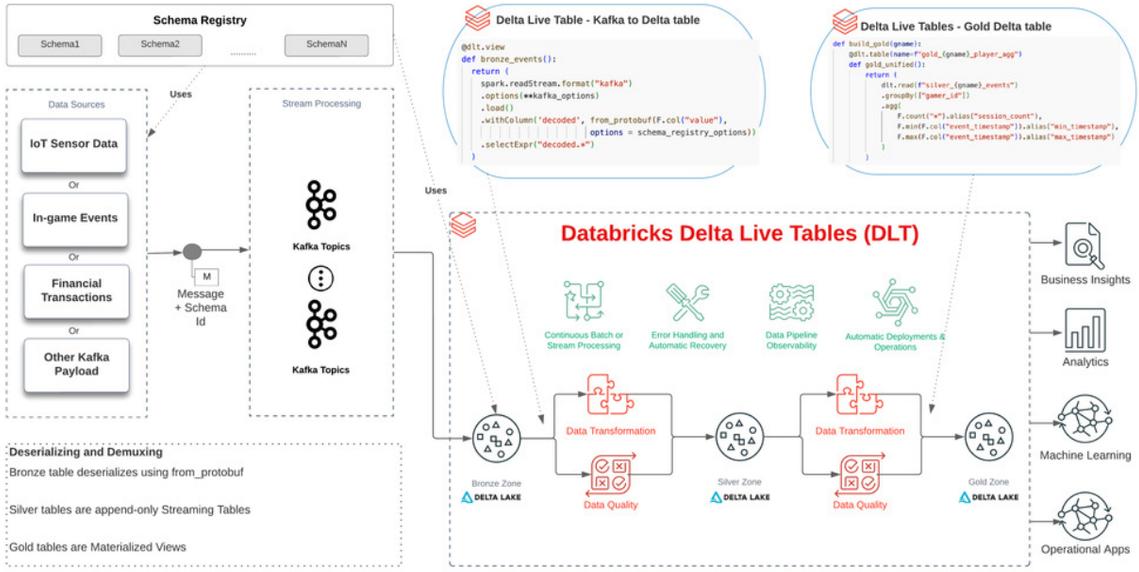
A Data Engineer's Guide to Optimized Streaming With Protobuf and Delta Live Tables

by Craig Lukasik

This article describes an example use case where events from multiple games stream through Kafka and terminate in Delta tables. The example illustrates how to use Delta Live Tables (DLT) to:

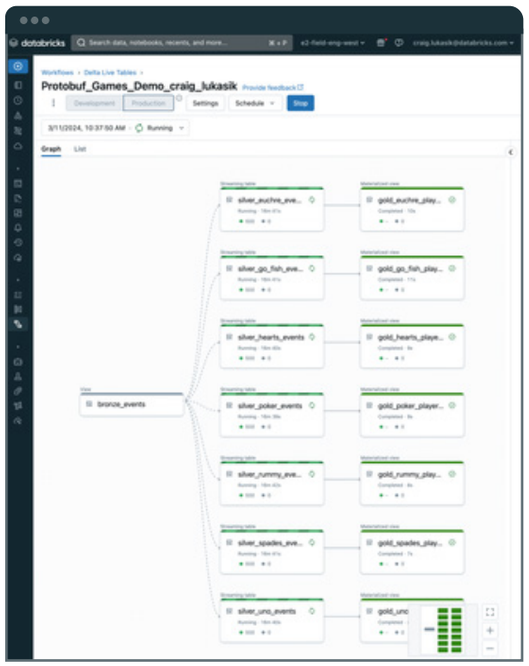
1. Stream from Kafka into a Bronze Delta table.
2. Consume streaming Protobuf messages with schemas managed by the Confluent Schema Registry, handling schema evolution gracefully.
3. Demultiplex (demux) messages into multiple game-specific, append-only Silver Streaming Tables. Demux indicates that a single stream is split or fanned out into separate streams.
4. Create Materialized Views to recalculate aggregate values periodically.

A high-level view of the system architecture is illustrated below.



First, let's look at the Delta Live Tables code for the example and the related pipeline DAG so that we can get a glimpse of the simplicity and power of the DLT framework.

```
The source bronze view, consuming the Kafk...

@dlt.view
def bronze_events():
  return (
    spark.readStream.format("kafka")
    .options(**kafka_options)
    .load()
    .withColumn('decoded', from_protobuf(F.col("value"),
                                         options = schema_registry_options))
    .selectExpr("decoded.*")
  )

Helper functions to define DLT tables

def build_silver(gname):
    @dlt.table(name=f"silver_{gname}_events")
    def gold_unified():
        return dlt.read_stream("bronze_events").where(F.col("game_name") == gname)

def build_gold(gname):
    @dlt.table(name=f"gold_{gname}_player_agg")
    def gold_unified():
        return (
            dlt.read(f"silver_{gname}_events")
            .groupBy(["gamer_id"])
            .agg(
                F.count("*").alias("session_count"),
                F.min(F.col("event_timestamp")).alias("min_timestamp"),
                F.max(F.col("event_timestamp")).alias("max_timestamp")
            )
        )

Silver Delta tables for each game. Notice the...

for game in GAMES_ARRAY:
    build_silver(game)
    build_gold(game)
```



On the left, we see the DLT Python code. On the right, we see the view and the tables created by the code. The bottom cell of the notebook on the left operates on a list of games (GAMES_ARRAY) to dynamically generate the fourteen target tables we see in the DAG.

Before we go deeper into the example code, let's take a step back and review streaming use cases and some streaming payload format options.

STREAMING OVERVIEW

Skip this section if you're familiar with streaming use cases, protobuf, the schema registry, and Delta Live Tables. In this article, we'll dive into a range of exciting topics.

- **Common streaming use cases:** Uncover the diverse streaming data applications in today's tech landscape.
- Protocol buffers (Protobuf): Learn why this fast and compact serialization format is a game-changer for data handling.
- Delta Live Tables (DLT): Discover how DLT pipelines offer a rich, feature-packed platform for your ETL (Extract, Transform, Load) needs.
- **Building a DLT pipeline**: A step-by-step guide on creating a DLT pipeline that seamlessly consumes Protobuf values from an Apache Kafka stream.
- **Utilizing the** Confluent Schema Registry: Understand how this tool is crucial for decoding binary message payloads effectively.
- Schema evolution **in DLT pipelines**: Navigate the complexities of schema evolution within the DLT pipeline framework when streaming protobuf messages with evolving schema.

COMMON STREAMING USE CASES

The Databricks Data Intelligence Platform is a comprehensive data-to-AI enterprise solution that combines data engineers, analysts, and data scientists on a single platform. Streaming workloads can power near real-time prescriptive and predictive analytics and automatically retrain Machine Learning (ML) models using Databricks built-in MLOps support. The models can be exposed as scalable, serverless REST endpoints, all within the Databricks platform.

The data comprising these streaming workloads may originate from various use cases:

STREAMING DATA	USE CASE
IoT sensors on manufacturing floor equipment	Generating predictive maintenance alerts and preemptive part ordering
Set-top box telemetry	Detecting network instability and dispatching service crews
Player metrics in a game	Calculating leader-board metrics and detecting cheat

Data in these scenarios is typically streamed through open source messaging systems, which manage the data transfer from producers to consumers. Apache Kafka stands out as a popular choice for handling such payloads. Confluent Kafka and AWS MSK provide robust Kafka solutions for those seeking managed services.

OPTIMIZING THE STREAMING PAYLOAD FORMAT

Databricks provides capabilities that help optimize the AI journey by unifying Business Analysis, Data Science, and Data Analysis activities in a single, governed platform. In your quest to optimize the end-to-end technology stack, a key focus is the serialization format of the message payload. This element is crucial for efficiency and performance. We'll specifically explore an optimized format developed by Google, known as protocol buffers (or "protobuf"), to understand how it enhances the technology stack.

WHAT MAKES PROTOBUF AN OPTIMIZED SERIALIZATION FORMAT?

Google enumerates the advantages of protocol buffers, including compact data storage, fast parsing, availability in many programming languages, and optimized functionality through automatically generated classes.

A key aspect of optimization usually involves using pre-compiled classes in the consumer and producer programs that a developer typically writes. In a nutshell, consumer and producer programs that leverage protobuf are "aware" of a message schema, and the binary payload of a protobuf message benefits from primitive data types and positioning within the binary message, removing the need for field markers or delimiters.

WHY IS PROTOBUF USUALLY PAINFUL TO WORK WITH?

Programs that leverage protobuf must work with classes or modules compiled using protoc (the protobuf compiler). The protoc compiler compiles those definitions into classes in various languages, including Java and Python. To learn more about how protocol buffers work, go here.

DATABRICKS MAKES WORKING WITH PROTOBUF EASY

Starting in Databricks Runtime 12.1, Databricks provides native support for serialization and deserialization between Apache Spark struct.... Protobuf support is implemented as an Apache Spark DataFrame transformation and can be used with Structured Streaming or for batch operations. It optionally integrates with the Confluent Schema Registry (a Databricks-exclusive feature).

Databricks makes it easy to work with protobuf because it handles the protobuf compilation under the hood for the developer. For instance, the data pipeline developer does not have to worry about installing protoc or using it to compile protocol definitions into Python classes.

EXPLORING PAYLOAD FORMATS FOR STREAMING IOT DATA

Before we proceed, it is worth mentioning that JSON or Avro may be suitable alternatives for streaming payloads. These formats offer benefits that, for some use cases, may outweigh protobuf. Let's quickly review these formats.

JSON

JSON is an excellent format for development because it is primarily human-readable. The other formats we'll explore are binary formats, which require tools to inspect the underlying data values. Unlike Avro and protobuf, however, the JSON document is stored as a large string (potentially compressed), meaning more bytes may be used than a value represents. Consider the short int value of 8. A short int requires two bytes. In JSON, you may have a document that looks like the following, and it will require several bytes (~30) for the associated key, quotes, etc.

```
{
  "my_short": 8
}
```

When we consider protobuf, we expect 2 bytes plus a few more for the overhead related to the positioning metadata.

JSON SUPPORT IN DATABRICKS

On the positive side, JSON documents have rich benefits when used with Databricks. Databricks Autoloader can easily transform JSON to a structured DataFrame while also providing built-in support for:

- Schema inference - when reading JSON into a DataFrame, you can supply a schema so that the target DataFrame or Delta table has the desired schema. Or you can let the engine infer the schema. Alternatively, schema hints can be supplied if you want a balance of those features.
- Schema evolution - Autoloader provides options for how a workload should adapt to changes in the schema of incoming files.

Consuming and processing JSON in Databricks is simple. To create a Spark DataFrame from JSON files can be as simple as this:

```
df = spark.read.format("json").load("example.json")
```

AVRO

Avro is an attractive serialization format because it is compact, encompasses schema information in the files themselves, and has built-in database support in Databricks that includes schema registry integration. This tutorial, co-authored by Databricks' Angela Chu, walks you through an example that leverages Confluent's Kafka and Schema Registry.

To explore an Avro-based dataset, it is as simple as working with JSON:

```
df = spark.read.format("avro").load("example.avro")
```

This datageeks.com article compares Avro and protobuf. It is worth a read if you are on the fence between Avro and protobuf. It describes protobuf as the "fastest amongst all.", so if speed outweighs other considerations, such as JSON and Avro's greater simplicity, protobuf may be the best choice for your use case.

EXAMPLE DEMUX PIPELINE

The source code for the end-to-end example is located on GitHub. The example includes a simulator (Producer), a notebook to install the Delta Live Tables pipeline (Install_DLT_Pipeline), and a Python notebook to process the data that is streaming through Kafka (DLT).

SCENARIO

Imagine a scenario where a video gaming company is streaming events from game consoles and phone-based games for a number of the games in its portfolio. Imagine the game event messages have a single schema that evolves (i.e., new fields are periodically added). Lastly, imagine that analysts want the data for each game to land in its own Delta Lake table. Some analysts and BI tools need pre-aggregated data, too.

Using DLT, our pipeline will create 1+2N tables:

- One table for the raw data (stored in the Bronze view).
- One Silver Streaming Table for each of the N games, with events streaming through the Bronze table.
- Each game will also have a Gold Delta table with aggregates based on the associated Silver table.

CODE WALKTHROUGH

BRONZE TABLE DEFINITION

We'll define the Bronze table (bronze_events) as a DLT view by using the @dlt.view annotation

```
1  import pyspark.sql.functions as F
2  from pyspark.sql.protobuf.functions import from_protobuf
3
4  @dlt.view
5  def bronze_events():
6    return (
7      spark.readStream.format("kafka")
8      .options(**kafka_options)
9      .load()
10     .withColumn('decoded', from_protobuf(F.col("value"), options = schema_registry_
   options))
11     .selectExpr("decoded.*")
12   )
```

The repo includes the source code that constructs values for kafka_options. These details are needed so the streaming Delta Live Table can consume messages from the Kafka topic and retrieve the schema from the Confluent Schema registry (via config values in schema_registry_options). This line of code is what manages the deserialization of the protobuf messages:

```
1  .withColumn('decoded', from_protobuf(F.col("value"), options = schema_registry_
   options))
2
```

The simplicity of transforming a DataFrame with protobuf payload is thanks to this function: from_protobuf (available in Databricks Runtime 12.1 and later). In this article, we don't cover to_protobuf, but the ease of use is the same. The schema_registry_options are used by the function to look up the schema from the Confluent Schema Registry.

Delta Live Tables is a declarative ETL framework that simplifies the development of data pipelines. So, suppose you are familiar with Apache Spark Structured Streaming. In that case, you may notice the absence of a checkpointLocation (which is required to track the stream's progress so that the stream can be stopped and started without duplicating or dropping data). The absence of the checkpointLocation is because Delta Live Tables manages this need out-of-the-box for you. Delta Live Tables also has other features that help make developers more agile and provide a common framework for ETL across the enterprise. Delta Live Tables Expectations, used for managing data quality, is one such feature.

SILVER TABLES

The following function creates a Silver Streaming Table for the given game name provided as a parameter:

```
def build_silver(gname):
    .table(name=f"silver_{gname}_events")
    def gold_unified():
        return dlt.read_stream("bronze_events").where(F.col("game_name") == gname)
```

Notice the use of the @dlt.table annotation. Thanks to this annotation, when build_silver is invoked for a given gname, a DLT table will be defined that depends on the source bronze_events table. We know that the tables created by this function will be Streaming Tables because of the use of dlt.read_stream.

GOLD TABLES

The following function creates a Gold Materialized View for the given game name provided as a parameter:

```
def build_gold(gname):
    .table(name=f"gold_{gname}_player_agg")
    def gold_unified():
        return (
            dlt.read(f"silver_{gname}_events")
            .groupBy(["gamer_id"])
            .agg(
                F.count("*").alias("session_count"),
                F.min(F.col("event_timestamp")).alias("min_timestamp"),
                F.max(F.col("event_timestamp")).alias("max_timestamp")
            )
        )
```

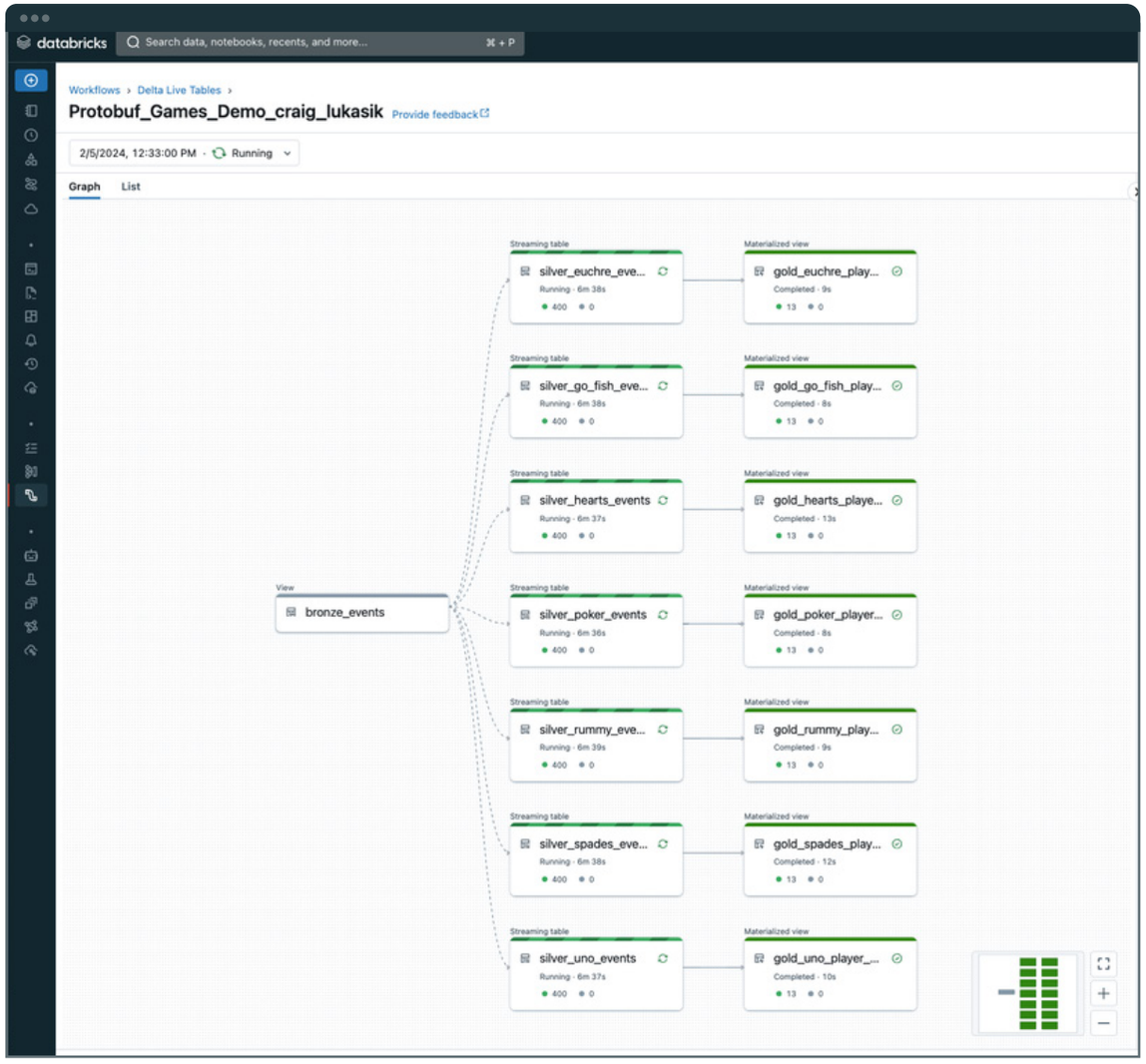
We know the resulting table will be a "Materialized View" because of the use of dlt.read. This is a simple Materialized View definition; it simply performs a count of source events along with min and max event times, grouped by gamer_id.

METADATA-DRIVEN TABLES

The previous two sections of this article defined functions for creating Silver (Streaming) Tables and Gold Materialized Views. The metadata-driven approach in the example code uses a pipeline input parameter to create N*2 target tables (one Silver table for each game and one aggregate Gold table for each game). This code drives the dynamic table creation using the aforementioned build_silver and build_gold functions:

```
1  GAMES_ARRAY = spark.conf.get("games").split(",")
2
3  for game in GAMES_ARRAY:
4      build_silver(game)
       build_gold(game)
```

At this point, you might have noticed that much of the control flow code data engineers often have to write is absent. This is because, as mentioned above, DLT is a declarative programming framework. It automatically detects dependencies and manages the pipeline's execution flow. Here's the DAG that DLT creates for the pipeline:



A note about aggregates in a streaming pipeline