# Chapter 5. Processing Incremental Data

In the previous chapters, we explored the fundamentals of processing data in groups or batches at once. However, when data is generated continuously, traditional batch processing approaches tend to become insufficient. In this chapter, we will explore the concepts and techniques for processing streaming data, including Spark Structured Streaming and incremental data ingestion from files. Moreover, we will discuss the concept of medallion architecture and how to build it under the stream processing model.

## Streaming Data with Apache Spark

Apache Spark provides robust support for processing streaming data, enabling you to efficiently perform real-time analytics. At the heart of this process is the concept of a data stream, which is the focus of processing. To effectively work with streaming data in Spark, let’s first understand what a data stream is and its characteristics.

### What Is a Data Stream?

A data stream represents an unbounded flow of data, often originating from various sources such as sensors, log files, or social media platforms. As new data is generated, it is appended to the stream, making it a dynamic and constantly changing dataset. Examples of data streams include the following:

*Social media feeds*
: Continuous streams of posts, each containing text, user information, and timestamps, that can be processed and analyzed to track trends, sentiments, or user behavior.

*Sensor readings*
: Temperature and humidity readings, or other metrics, from a network of sensors in a smart building, used to optimize energy consumption.

*Log data*
: A stream of log messages from a server, containing system events and error messages, used to monitor system performance or detect security threats.

Processing data streams present a unique set of challenges due to their dynamic and ever-growing nature. To handle such continuous flows of information, there are typically two primary approaches:

*Recompute*

In this classical approach, each time new data arrives, the entire dataset is reprocessed to incorporate the new information. While this method ensures accuracy, it can be computationally intensive and time-consuming, especially for large datasets.

*Incremental processing*

Alternatively, incremental processing involves developing custom logic to identify and capture only the new data that has been added since the last update. This approach reduces processing overhead by focusing solely on the changes, thereby improving efficiency.

One powerful tool for incremental processing of data streams is Spark Structured Streaming, which is part of Apache Spark.

## Spark Structured Streaming

Spark Structured Streaming is a scalable stream processing engine that revolutionizes the way data streams are processed and queried. It enables querying of infinite data sources, automatically detecting new data as it arrives and persisting results incrementally into target data sinks, as illustrated in Figure 5-1. A sink is often a durable storage system such as files or tables that serves as the destination for the processed data.

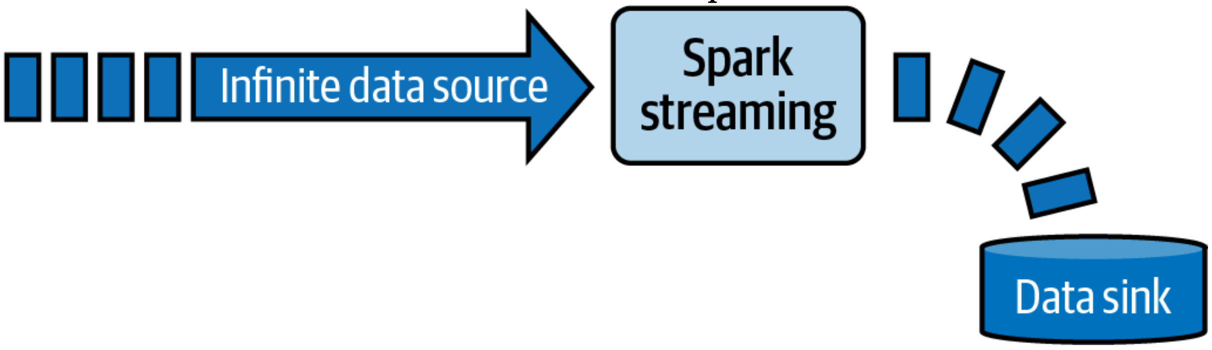


Figure 5-1. Spark Structured Streaming

In Structured Streaming, the key idea is to handle live data streams as unbounded, continuously growing tables, where each incoming data item is appended as a new row, as illustrated in Figure 5-2. This design allows you to apply familiar SQL and DataFrame operations on streaming data in the same way you would with batch data. By unifying batch and streaming operations, Structured Streaming eliminates the need for separate technology stacks and facilitates the migration of your existing batch Spark jobs to streaming jobs.

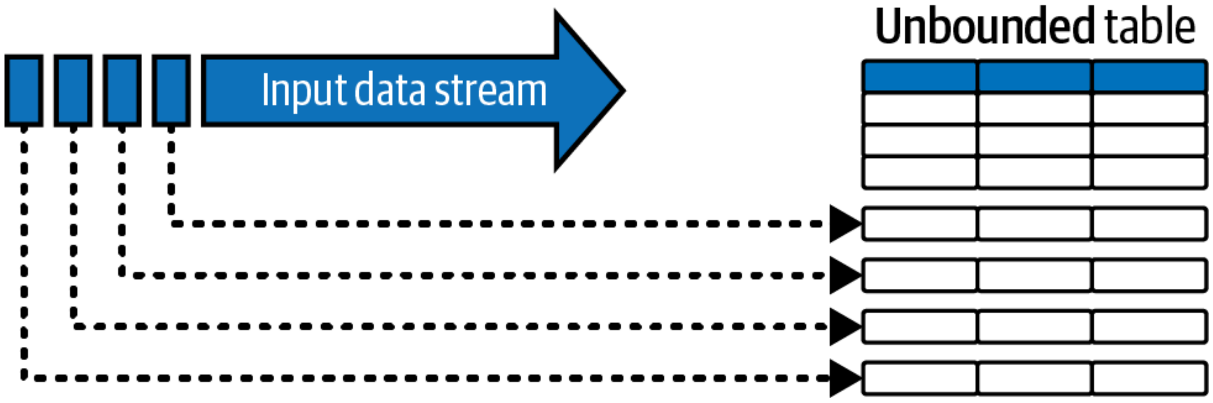


Figure 5-2. Fundamental concept of Spark Structured Streaming (image adapted from https://spark.apache.org)

### The append-only requirement of streaming sources

One fundamental prerequisite for a data source to be considered valid for streaming is that it must adhere to the append-only requirement in Structured Streaming. This condition implies that data can only be added to the source, and existing data cannot be modified. If a data source allows data to be updated, deleted, or overwritten, it is then considered no longer streamable.

Therefore, it is essential to ensure that your data sources conform to this requirement in order to take advantage of the benefits of streaming data processing.

### Delta Lake as streaming source

Spark Structured Streaming seamlessly integrates with various data sources, including directories of files, messaging systems like Kafka, and Delta Lake tables as well. Delta Lake is well-integrated with Spark Structured Streaming using the DataStreamReader and DataStreamWriter APIs in PySpark.

#### *DataStreamReader*

In Python, the `spark.readStream` method allows you to query a Delta Lake table as a streaming source. This functionality enables processing both existing data in the table and any new data that arrives subsequently. The result is a “streaming” DataFrame, which allows for applying transformations just like one would on a static DataFrame:

```
streamDF = spark.readStream.table("source_table")
```

#### *DataStreamWriter*

Once the necessary transformations have been applied, the results of the streaming DataFrame can be persisted using its `writeStream` method:

```
streamDF.writeStream.table("target_table")
```
This method enables configuring various output options to store the processed data into durable storage. Let's explore the following example, where we have two Delta Lake tables, `Table_1` and `Table_2`. The goal is to continuously stream data from `Table_1` to `Table_2`, appending new records into `Table_2` every two minutes, as illustrated in Figure 5-3.

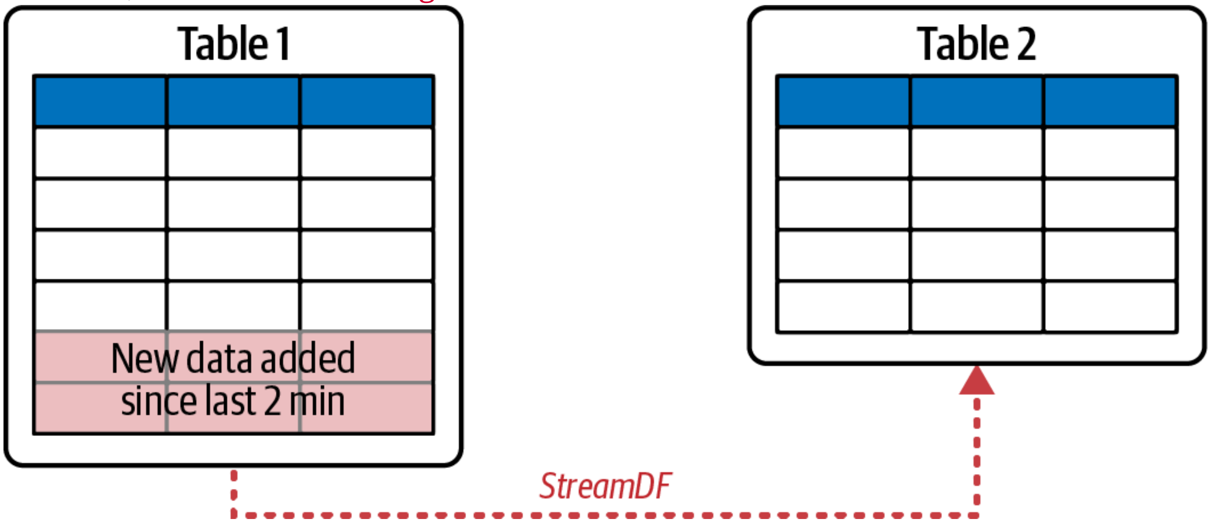


Figure 5-3. Streaming data between two Delta Lake tables

To achieve this, we use the following Python code. This code sets up a Structured Streaming job in Spark that continuously monitors `Table_1` for new data, processes it at regular intervals of two minutes, and appends the new records to `Table_2`:
```
streamDF = spark.readStream
                  .table("Table_1")

streamDF.writeStream
         .trigger(processingTime="2 minutes")
         .outputMode("append")
         .option("checkpointLocation", "/path")
         .table("Table_2")
```
In this code snippet, we start by defining a streaming DataFrame `streamDF` against the Delta table `Table_1` using the `spark.readStream` method. Whenever a new version of the table is written, a new *micro-batch* containing the new data will come in through this `readStream`. Next, we use the `writeStream` method to define the streaming write operation on the `streamDF`. Here, we specify the processing trigger interval using the trigger function, indicating that Spark should check for new data every two minutes. This means that the streaming job will be triggered at regular intervals of two minutes to process any new incoming data in the source.
We then set the output mode to "append" using the `outputMode` function. This mode ensures that only newly received records since the last trigger will be appended to the output sink, which in this case is the Delta table `Table_2`. Additionally, we specify the checkpoint location using the `checkpointLocation` option. Spark uses checkpoints to store metadata about

the streaming job, including the current state and progress. By providing a checkpoint location, Spark can recover the streaming job from failures and maintain its state across restarts.

## Streaming Query Configurations

Now, let’s examine the configurations of DataStreamWriter in detail.

### Trigger Intervals

When setting up a streaming write operation, the trigger method defines how often the system should process incoming data. This timing mechanism is referred to as the trigger interval. There are two primary trigger modes: continuous and triggered, as illustrated in Table 5-1.

| Mode | Usage | Behavior |
|---|---|---|
| Continuous | `.trigger(processingTime="5 minutes")` | Processes data at fixed intervals (e.g., every 5 minutes). Default interval: 500ms. |
| Triggered | `#deprecated .trigger(once=True)` | Processes all available data in a single micro-batch, then stops automatically. |
|  | `.trigger(availableNow=True)` | Processes all available data in multiple micro-batches, then stops automatically. |

Table 5-1. Trigger intervals of DataStreamWriter

Let’s dive deeper to gain a comprehensive understanding of these two modes.

#### *Continuous mode: Near-real-time processing*

In this mode, the streaming query will continuously run to process data in micro-batches at regular intervals. By default, if no specific trigger interval is provided, the data will be processed every half a second. This is equivalent to using the option `processingTime="500ms"`. Alternatively, you have the flexibility to specify another fixed interval according to your requirements. For instance, you might opt to process the data at a specified interval, such as every five minutes, by using the option `processingTime="5 minutes"`. This mode ensures a continuous flow of data, enabling near-real-time data processing.

### *Triggered mode: Incremental batch processing*

In contrast to continuous mode, the triggered mode offers a batch-oriented approach known as incremental batch processing. In this mode, the streaming query processes all available data since the last trigger and then stops automatically. This mode is suited for scenarios where data arrival is not constant, eliminating the need for continuously running resources. Under the triggered mode, two options are available: `Once` and `availableNow`:

*`Trigger.Once`*

With this option, the stream processes the currently available data, all at once, in a single micro-batch. However, this can introduce challenges related to scalability when dealing with large volumes of data, as it may lead to out-of-memory (OOM) errors.

*`Trigger.availableNow`*

Similarly, the `availableNow` option also facilitates batch processing of all currently available data. However, it addresses scalability concerns by allowing data to be processed in multiple micro-batches until completion. This option offers flexibility in handling large data batches while ensuring efficient resource utilization.

**NOTE**

Since Databricks Runtime 11.3 LTS, the `Trigger.Once` setting has been deprecated. However, it's possible that you may encounter references to it in the current exam version or in older documentation. Databricks now recommends using `Trigger.AvailableNow` for all incremental batch processing workloads.

## Output Modes

When writing streaming data, you can specify the output mode to define how the data is written to the target. There are primarily two output modes available: append mode and complete mode, as illustrated in Table 5-2.

| Mode | Usage | Behavior |
|---|---|---|
| Append (default) | `.outputMode("append")` | Only newly received rows are appended to the target table with each batch. |
| Complete | `.outputMode("complete")` | The target table is overwritten with each batch. |

Table 5-2. Output modes of `DataStreamWriter`

*Append mode*

Append mode is the default output mode if no specific mode is provided. It appends only new rows that have been received since the last trigger to the target table. This mode is suitable for scenarios where the target sink needs to maintain a continuously growing dataset based on the incoming streaming data.

*Complete mode*

Complete mode recomputes and rewrites the entire results to the sink every time a write is triggered. It replaces the entire contents of the output sink with the latest computed results with each batch. This mode is commonly used for updating summary tables with the latest aggregates.

### Checkpointing

Checkpointing is a mechanism for saving the progress information of the streaming query. The checkpoints are stored in a reliable storage system, such as the DBFS or cloud storage like Amazon S3 or Azure Storage. This approach ensures that if the streaming job crashes or needs to be restarted, it can resume processing from the last checkpointed state rather than starting from scratch.

One important aspect to note about checkpoints in Apache Spark is that they cannot be shared between multiple streaming jobs. Each streaming write operation requires its own separate checkpoint location. This separation ensures that each streaming application maintains its own processing guarantees and doesn't interfere with or rely on the checkpoints of other jobs.

## Structured Streaming Guarantees

Spark Structured Streaming offers, primarily, two guarantees to ensure end-to-end reliable and fault-tolerant stream processing: fault recovery and exactly-once semantics.

### Fault recovery

In case of failures, such as node crashes or network issues, the streaming engine is capable of resuming processing from where it left off. This is achieved through the combination of checkpointing and a mechanism called write-ahead logs. They enable capturing the offset range corresponding to the data being processed during every trigger, which makes it possible to recover from failures without any data loss.

It’s important to note that this guarantee mainly depends on the repeatability of the data sources. Data sources such as cloud-based object storage or pub/sub messaging services are typically repeatable, meaning that the same data can be read multiple times if needed. This allows the streaming engine to restart or reprocess the data under any failure condition.

#### Exactly-once semantics

Structured Streaming also guarantees that each record in the stream will be processed exactly once, even in the event of failures and retries. This is ensured by the implementation of idempotent streaming sinks. Idempotency means that if multiple writes occur for the same entities, no duplicates will be written to the sink. It relies on the offset of the entities as a unique identifier to recognize any duplicates and ignore them.

In essence, by accurately tracking offsets from replayable sources and leveraging idempotent sinks, Structured Streaming ensures reliable end-to-end processing, without any risk of data loss or duplication, even in the presence of failures.

### Unsupported operations

As discussed earlier, infinite data sources are viewed as unbounded tables in Structured Streaming. While most operations are identical to those of batch processing, there are certain operations that are not supported due to the nature of streaming data. Operations such as sorting and deduplication introduce complexities in a streaming context and may not be directly applicable or feasible.

While a full discussion of these limitations is beyond the scope of this Associate-level certification, it’s essential to know that there are alternative mechanisms to address similar requirements. For example, you can use advanced streaming techniques like windowing and watermarking for performing such operations over specific time windows. A detailed understanding of these techniques is typically expected at a more advanced level, particularly for the Databricks Data Engineer Professional certification.

## Implementing Structured Streaming

Let's delve into the practical implementation of Spark Structured Streaming for enabling incremental data processing. We will continue using our online school dataset, consisting of three tables: `students`, `enrollments`, and `courses`, as illustrated in Figure 5-4.

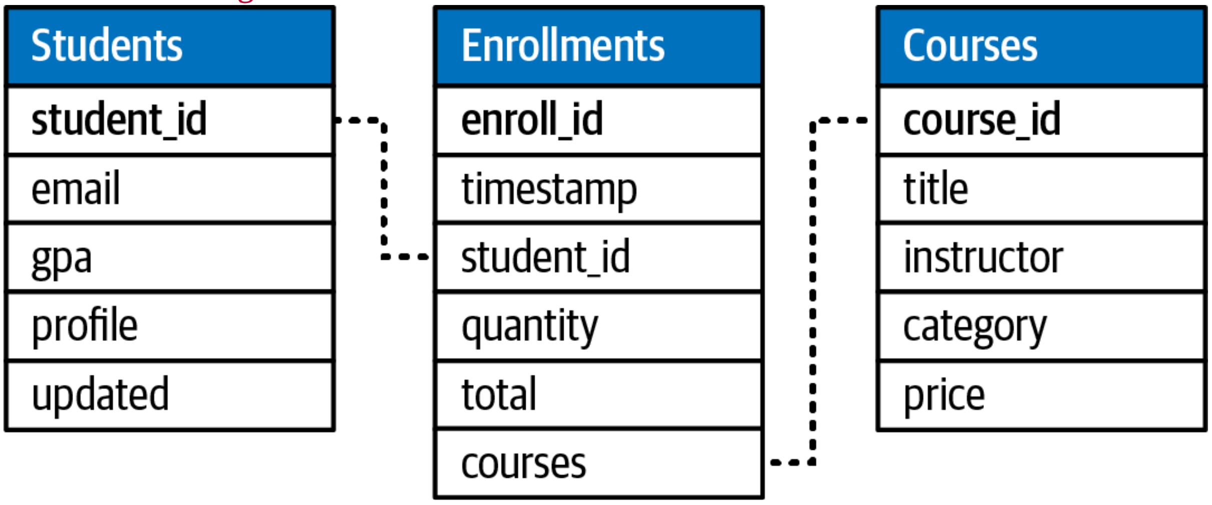


Figure 5-4. Entity-relationship diagram of the online school dataset

In this demonstration, we will use a new Python notebook titled "5.1 - Structured Streaming." We begin by running the "School-Setup" helper notebook to prepare our environment:

```
%run ../Includes/School-Setup
```

Structured Streaming provides high-level APIs in both SQL and Python for manipulating streaming data. However, regardless of the chosen language, the initial step always involves using the `spark.readStream` method from the PySpark API. This is the reason behind our utilization of a Python notebook in this context. The `spark.readStream` method allows you to query a Delta table as a streaming source and create a streaming DataFrame accordingly:

```
stream_df = spark.readStream.table("courses")
```

Once the streaming DataFrame is created, you can apply a wide range of transformations and operations to manipulate and analyze the streaming data. These transformations can be expressed in either SQL or Python syntax.

## Streaming Data Manipulations in SQL

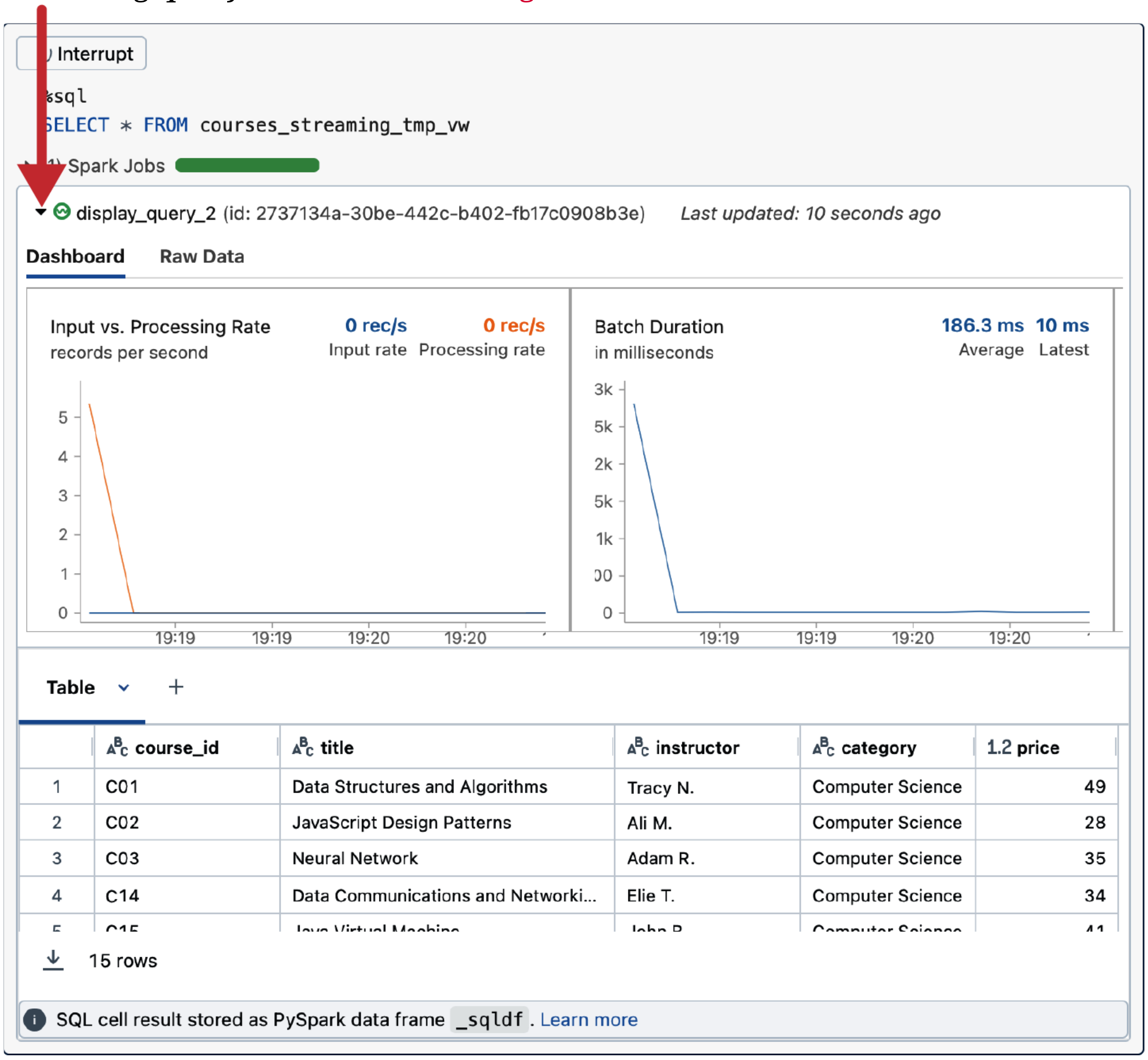
To begin manipulating streaming data using SQL, it is essential to convert the streaming DataFrame into a format that SQL can interpret and query. This can be achieved by registering a temporary view from the streaming DataFrame using the `createOrReplaceTempView` function:

```
stream_df.createOrReplaceTempView("courses_streaming_tmp_vw")
```

Creating a temporary view against a streaming DataFrame results in a `streaming` temporary view. This allows you to apply most SQL transformations on streaming data just like you would with static data. You can query this streaming temporary view using a standard `SELECT` statement, as shown here:

```
%sql
SELECT * FROM courses_streaming_tmp_vw
```

This query does not behave like a typical SQL query. Instead of executing once and returning a set of results, it initiates a continuous stream that runs indefinitely. As new data arrives in the source, it appears in the query results in near real time. To facilitate performance monitoring of such streams, Databricks Notebooks provide an interactive dashboard associated with the streaming query, as illustrated in Figure 5-5.



Interrupt
(1) Spark Jobs
display_query_2 (id: 2737134a-30be-442c-b402-fb17c0908b3e)
Last updated: 10 seconds ago
Dashboard
Raw Data
Input vs. Processing Rate
records per second
0 rec/s
Input rate
0 rec/s
Processing rate
Batch Duration
in milliseconds
186.3 ms
Average
10 ms
Latest
Table
15 rows
SQL cell result stored as PySpark data frame _sqldf. Learn more

```
%sql
SELECT * FROM courses_streaming_tmp_vw
```

| | course_id | title | instructor | category | price |
|---|---|---|---|---|---|
| 1 | C01 | Data Structures and Algorithms | Tracy N. | Computer Science | 49 |
| 2 | C02 | JavaScript Design Patterns | Ali M. | Computer Science | 28 |
| 3 | C03 | Neural Network | Adam R. | Computer Science | 35 |
| 4 | C14 | Data Communications and Networki... | Elie T. | Computer Science | 34 |
| 5 | C15 | Java Virtual Machine | John P. | Computer Science | 41 |

Figure 5-5. Streaming query results

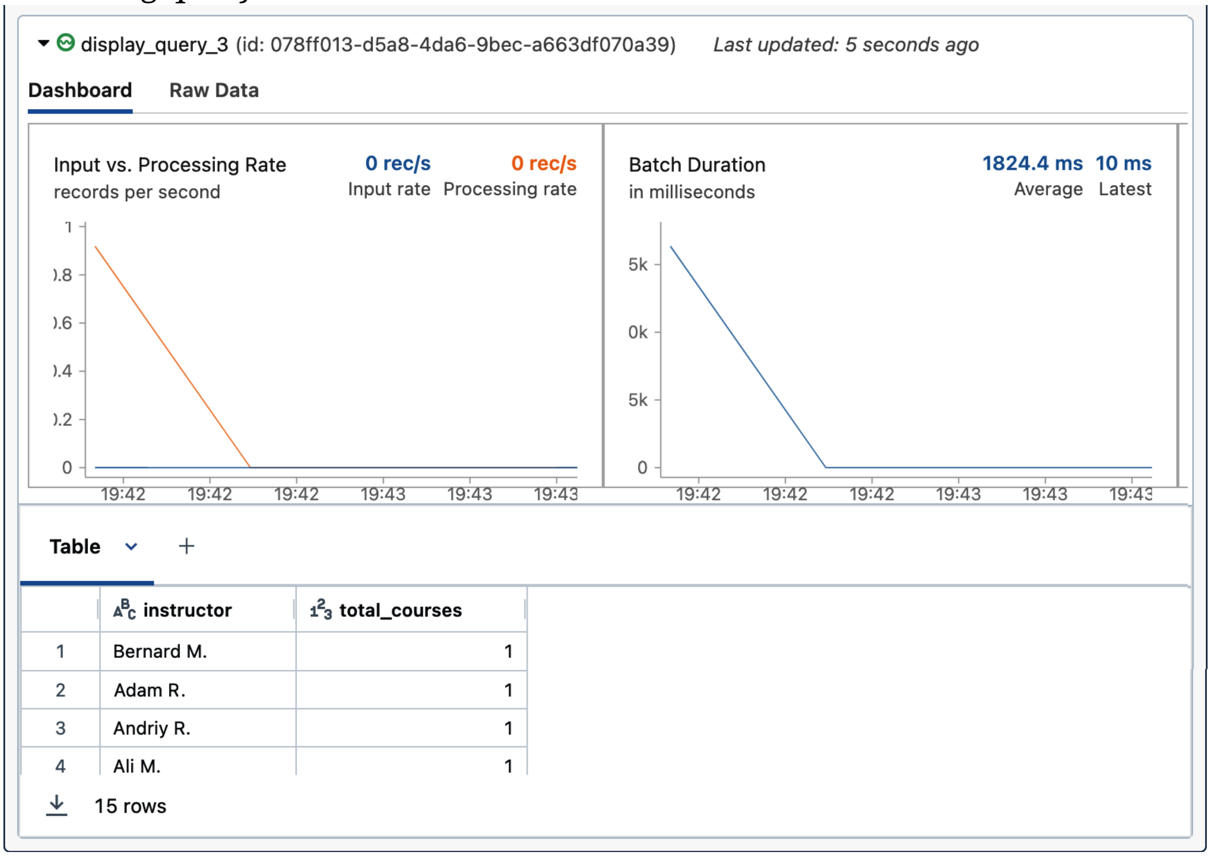
In practice, we don't typically display the results of streaming queries in a notebook unless there is a need to inspect them during development. To stop an active streaming query, you can simply click Interrupt at the top of the cell.

## Applying transformations

On a streaming temporary view, you can apply various transformations and operations. For instance, you can perform aggregations such as as counting occurrences within the streaming data:

```
%sql
SELECT instructor, count(course_id) AS total_courses
FROM courses_streaming_tmp_vw
GROUP BY instructor
```

Because we are querying a streaming object, this aggregation becomes a *stateful* streaming query that executes continuously and updates dynamically to reflect any changes in the source. Figure 5-6 displays the output of this streaming query.



| | instructor | total_courses |
|---|---|---|
| 1 | Bernard M. | 1 |
| 2 | Adam R. | 1 |
| 3 | Andriy R. | 1 |
| 4 | Ali M. | 1 |

Figure 5-6. Streaming aggregation results

It's important to understand that at this stage, none of the records is stored anywhere; they are simply being displayed in the current notebook environment. In the following discussion, we will explore how to persist them to a durable storage. However, before proceeding, let us stop this active streaming query.

Remember, when working with streaming data, certain SQL operations are not directly supported. For example, attempting to sort our streaming data based on a given column will lead to an error:

```
%sql
SELECT *
FROM courses_streaming_tmp_vw
ORDER BY instructor
AnalysisException: Sorting is not supported on streaming DataFrames/Datasets,
unless it is on aggregated DataFrame/Dataset in Complete output mode; line 3
pos 1;
```

Executing this command results in an exception, clearly indicating that the sorting operation is not supported on all streaming datasets. As mentioned earlier, more advanced techniques like windowing and watermarking can be used to overcome such limitations. However, they are considered beyond the scope of this book.

## Persisting streaming data

Persisting streaming data to a durable storage involves returning our logic back to the PySpark DataFrame API. For this, we begin by defining a new temporary view to encapsulate our desired output:

```
%sql
CREATE OR REPLACE TEMP VIEW instructor_counts_tmp_vw AS (
 SELECT instructor, count(course_id) AS total_courses
 FROM courses_streaming_tmp_vw
 GROUP BY instructor
)
```

With this SQL statement, we are creating another temporary view to hold the aggregated data. It's considered a "streaming" temporary view since it is derived from a query against a streaming object, specifically against our `courses_streaming_tmp_vw` view.

Once the streaming temporary view is created, we can access the output data using the PySpark DataFrame API. In the following snippet, the `spark.table` function loads the data from our streaming temporary view into a *streaming* DataFrame:

```
result_stream_df = spark.table("instructor_counts_tmp_vw")
```

It's essential to understand that Spark differentiates between streaming and static DataFrames. Consequently, when loading data from a streaming object, it's interpreted as a streaming DataFrame, while loading data from a static object yields a static DataFrame. This highlights the importance of using `spark.readStream` from the beginning (instead of `spark.read`) to support later incremental writing.

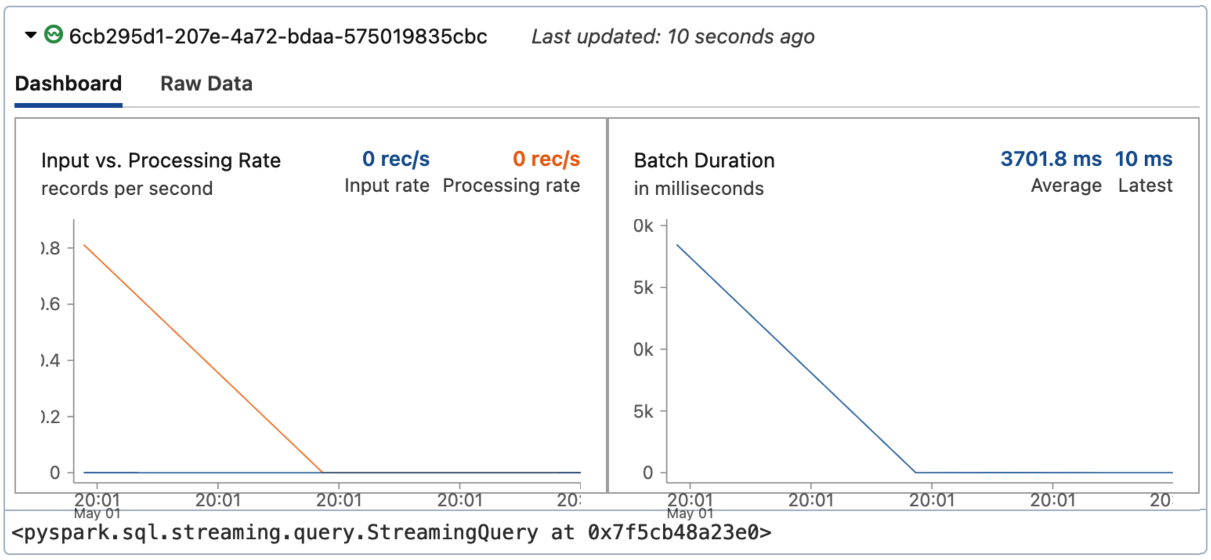
Now that we have our streaming DataFrame in place, we can proceed to persist the results to a Delta table using the `writeStream` method. This method enables configuring the output with several parameters, such as trigger intervals, output modes, and specifying a checkpoint location:

```
(result_stream_df.writeStream
                .trigger(processingTime='3 seconds')
                .outputMode("complete")
```

```
                        .option("checkpointLocation",
                                   "dbfs:/mnt/DEA-Book/checkpoints/instructor_counts")
                        .table("instructor_counts")
)
```

In this configuration, the trigger interval is set to three seconds, meaning the stream will attempt an update every three seconds by checking the source for new data. The output mode is specified as "complete," indicating that the entire target table should be overwritten with the new calculations during each trigger interval. Additionally, the checkpoint location is provided to track the progress of the stream processing. Lastly, the target table is set to `instructor_counts`.

Executing this command initiates a streaming query, continuously updating the target table as new data arrives. Figure 5-7 visualizes this process through its interactive dashboard.



```
<pyspark.sql.streaming.query.StreamingQuery at 0x7f5cb48a23e0>
```

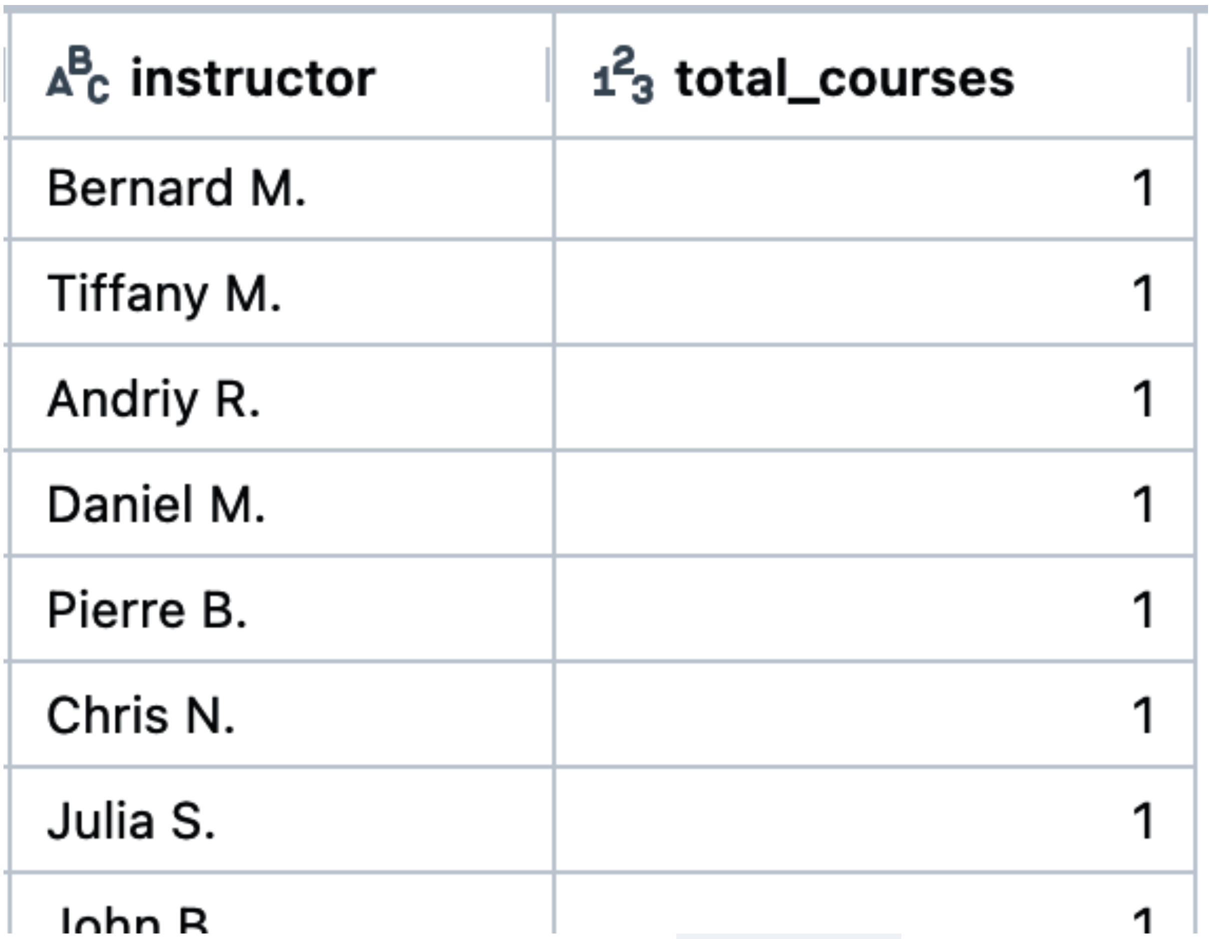
Figure 5-7. Streaming write operation

From this dashboard, we can observe a noticeable spike, indicating that our data has been processed. Subsequently, we can proceed to query the target table to validate the results:

```
%sql
SELECT * FROM instructor_counts
```

It's important to note that directly querying the target table does not trigger a streaming query. It's simply a normal, static table, rather than a streaming DataFrame.

Figure 5-8 displays the result of querying the `instructor_counts` table, confirming that the data has been written successfully. This result shows that each instructor currently teaches only one course.



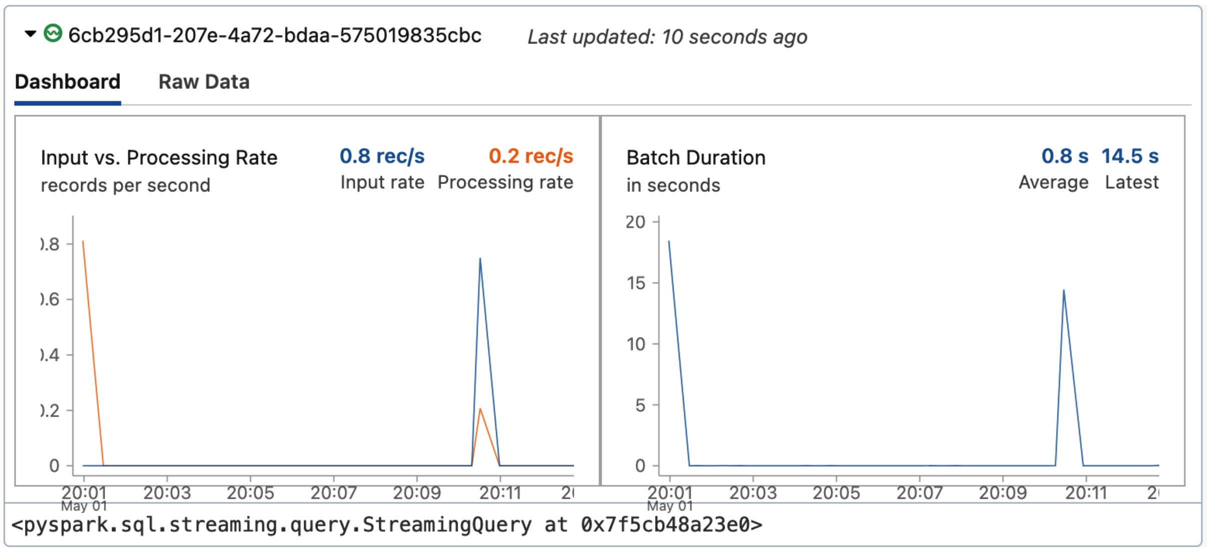
| instructor | total_courses |
|---|---|
| Bernard M. | 1 |
| Tiffany M. | 1 |
| Andriy R. | 1 |
| Daniel M. | 1 |
| Pierre B. | 1 |
| Chris N. | 1 |
| Julia S. | 1 |
| John B | 1 |

Figure 5-8. The result of querying the `instructor_counts` table

Meanwhile, the streaming write query remains active, waiting for new data to arrive in the source. To illustrate this, let us add new data to our source table, the `courses` table:

```
%sql
INSERT INTO courses
values ("C16", "Generative AI", "Pierre B.", "Computer Science", 25),
        ("C17", "Embedded Systems", "Julia S.", "Computer Science", 30),
        ("C18", "Virtual Reality", "Bernard M.", "Computer Science", 35)
```

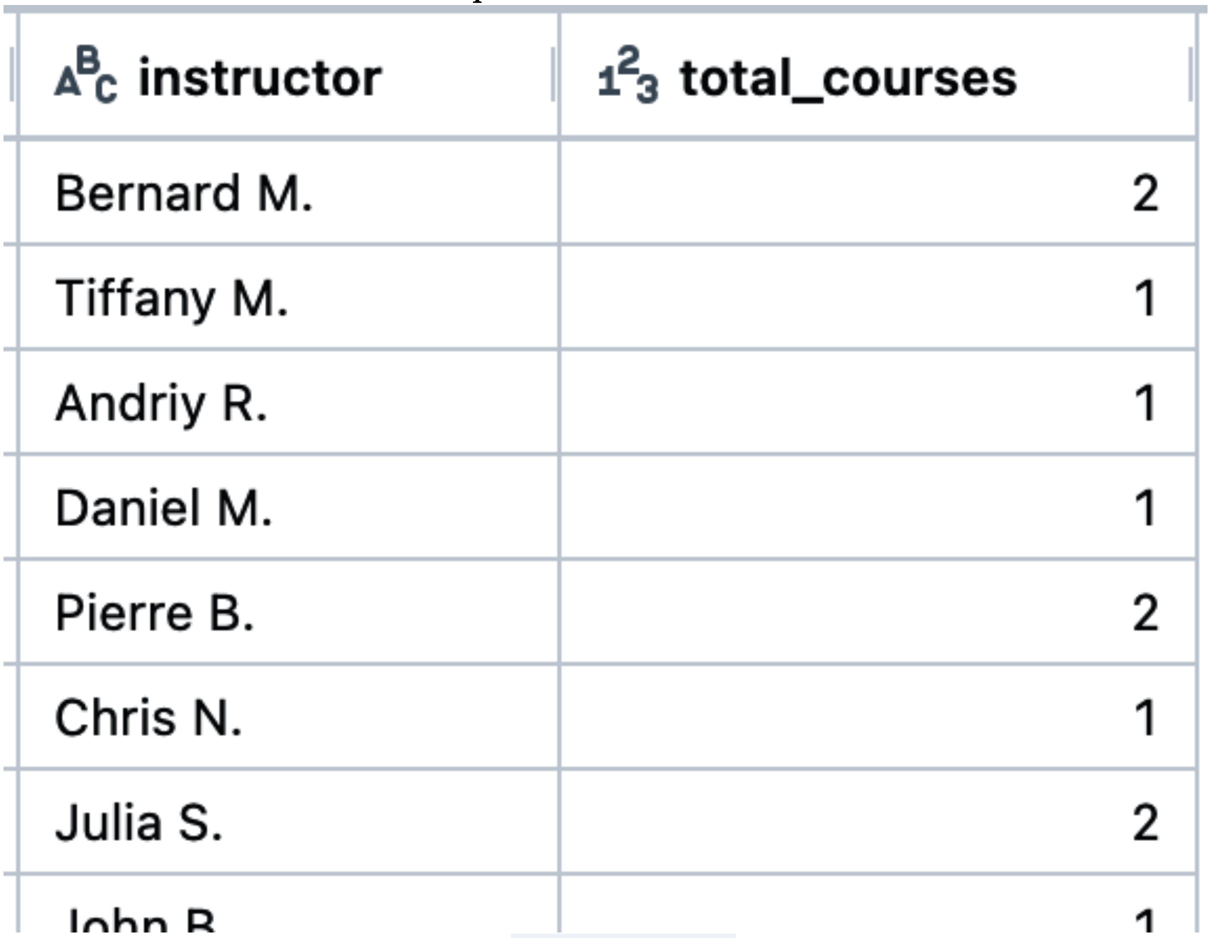
Upon executing this command, you can observe the processing of this batch of data using the dashboard of our streaming query, as shown in Figure 5-9.



```
<pyspark.sql.streaming.query.StreamingQuery at 0x7f5cb48a23e0>
```

Figure 5-9. Processing the new streaming data

Subsequently querying our target table reveals updated course counts for each instructor. As illustrated in Figure 5-10, some instructors' course counts have increased as more records are processed.



| instructor | total_courses |
|---|---|
| Bernard M. | 2 |
| Tiffany M. | 1 |
| Andriy R. | 1 |
| Daniel M. | 1 |
| Pierre B. | 2 |
| Chris N. | 1 |
| Julia S. | 2 |
| John B | 1 |

Figure 5-10. The result of querying the `instructor_counts` table after processing the new data

Now, let's explore another scenario to demonstrate incremental batch processing. However, before proceeding, let's stop our previous streaming write query. In a development environment, it is a good practice to stop active streams to prevent them from running indefinitely. Failing to do so can lead to unnecessary costs and resource consumption, as the cluster will not be able to auto-terminate if the process remains active.

In our next scenario, we will introduce a set of courses taught by new instructors and incorporate them into our source table:

```
%sql
INSERT INTO courses
values ("C19", "Compiler Design", "Sophie B.", "Computer Science", 25),
       ("C20", "Signal Processing", "Sam M.", "Computer Science", 30),
       ("C21", "Operating Systems", "Mark H.", "Computer Science", 35)
```

In this scenario, we will modify the trigger method to change our query from a continuous mode, executed every three seconds, to a triggered mode.  We accomplish this using the `availableNow` trigger option:

```
(result_stream_df.writeStream
                      .trigger(availableNow=True)
                      .outputMode("complete")
                      .option("checkpointLocation",
                              "dbfs:/mnt/DEA-Book/checkpoints/instructor_counts")
                      .table("instructor_counts")
                      .awaitTermination()
)
```

With the `availableNow` trigger option, the query will process all newly available data at the time of the read and automatically stop upon completion. In this case, we can optionally use the `awaitTermination` method to halt execution of other cells in the notebooks until the incremental batch write finishes successfully.

By running this command, you can observe that the streaming query was operated in a batch mode. It stopped automatically after processing the three recently added records. To confirm this, you can query the target table again to see that there are now 18 instructors instead of the previous 15.

## Streaming Data Manipulations in Python

Manipulating streaming data in Python syntax is straightforward; there is no need for any temporary object or view. You can apply your data processing directly on the streaming DataFrame using the PySpark DataFrame API:

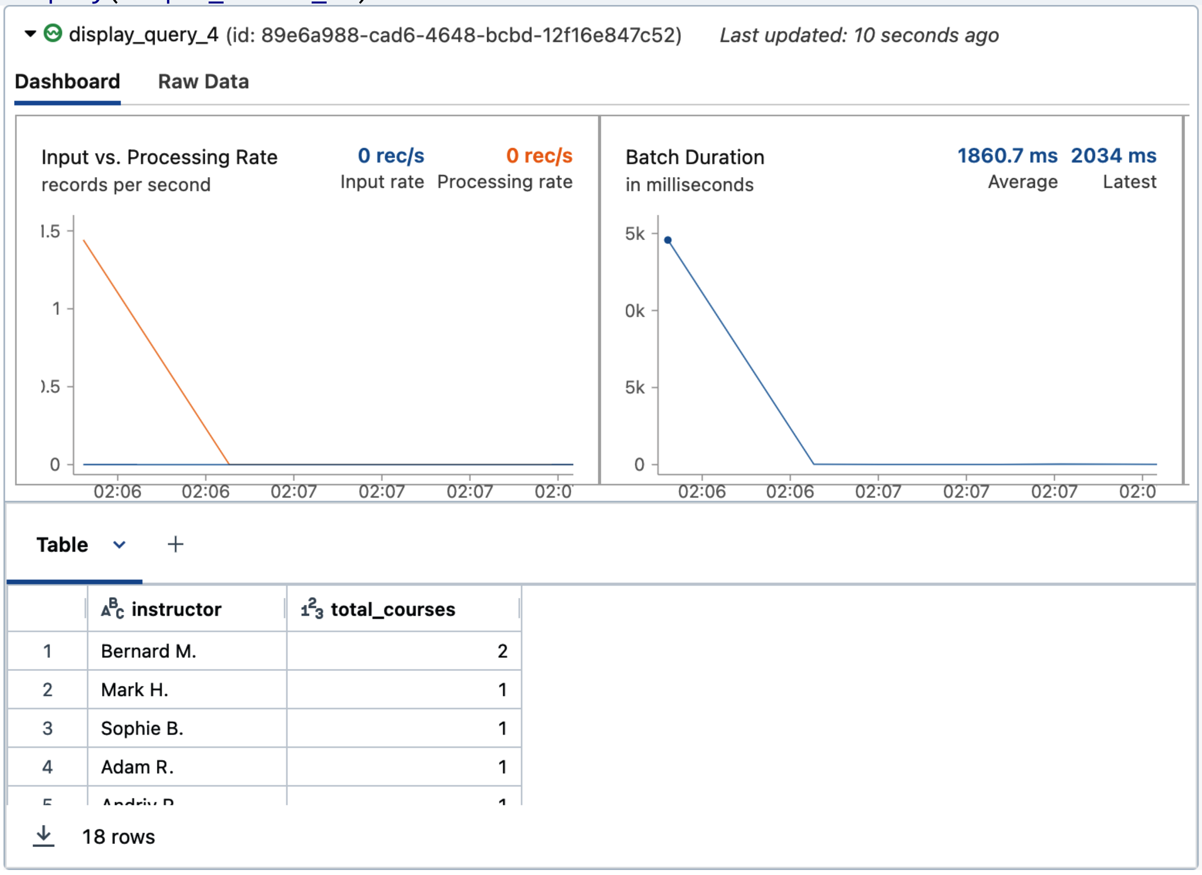
```
import pyspark.sql.functions as F

output_stream_df = (stream_df.groupBy("instructor")
                             .agg(F.count("course_id").alias("total_courses")))
```

In this snippet, we are performing the same aggregation operation previously executed using SQL syntax, but now using PySpark. We group our `stream_df` based on the `instructor` column and apply the count aggregation function to the `course_id` column. It's worth mentioning that streaming

DataFrames, like static DataFrames, are immutable. This means that when you apply transformations to a DataFrame, it always creates a new DataFrame and leaves the original unchanged. In our case, this creates a new streaming DataFrame named `output_stream_df`.
At this point, the output streaming DataFrame has been created, but the stream itself is not yet active. This means that Spark hasn't started processing the input data. To activate the stream, we need to perform an action, such as writing or displaying the data. In Databricks notebooks, you can call the display function on a streaming DataFrame to display the streaming data, as illustrated in Figure 5-11:

```
display(output_stream_df)
```



|  | instructor | total_courses |
|---|---|---|
| 1 | Bernard M. | 2 |
| 2 | Mark H. | 1 |
| 3 | Sophie B. | 1 |
| 4 | Adam R. | 1 |
| 5 | Andriy P. | 1 |

Figure 5-11. Displaying the streaming DataFrame

We can now stop this data stream and examine how to persist the results.

To persist these results to durable storage, we simply use the `writeStream` method directly on the streaming DataFrame:

```
(output_stream_df.writeStream
                    .trigger(availableNow=True)
                    .outputMode("complete")
                    .option("checkpointLocation",
                      "dbfs:/mnt/DEA-Book/checkpoints/instructor_counts_py")
                    .table("instructor_counts_py")
                    .awaitTermination()
)
```

It's essential to note that we are using a different checkpoint location for this new streaming query. Remember, each stream requires its own separate checkpoint location to ensure processing guarantees.

Once the streaming write is completed, you can query the resulting table directly:

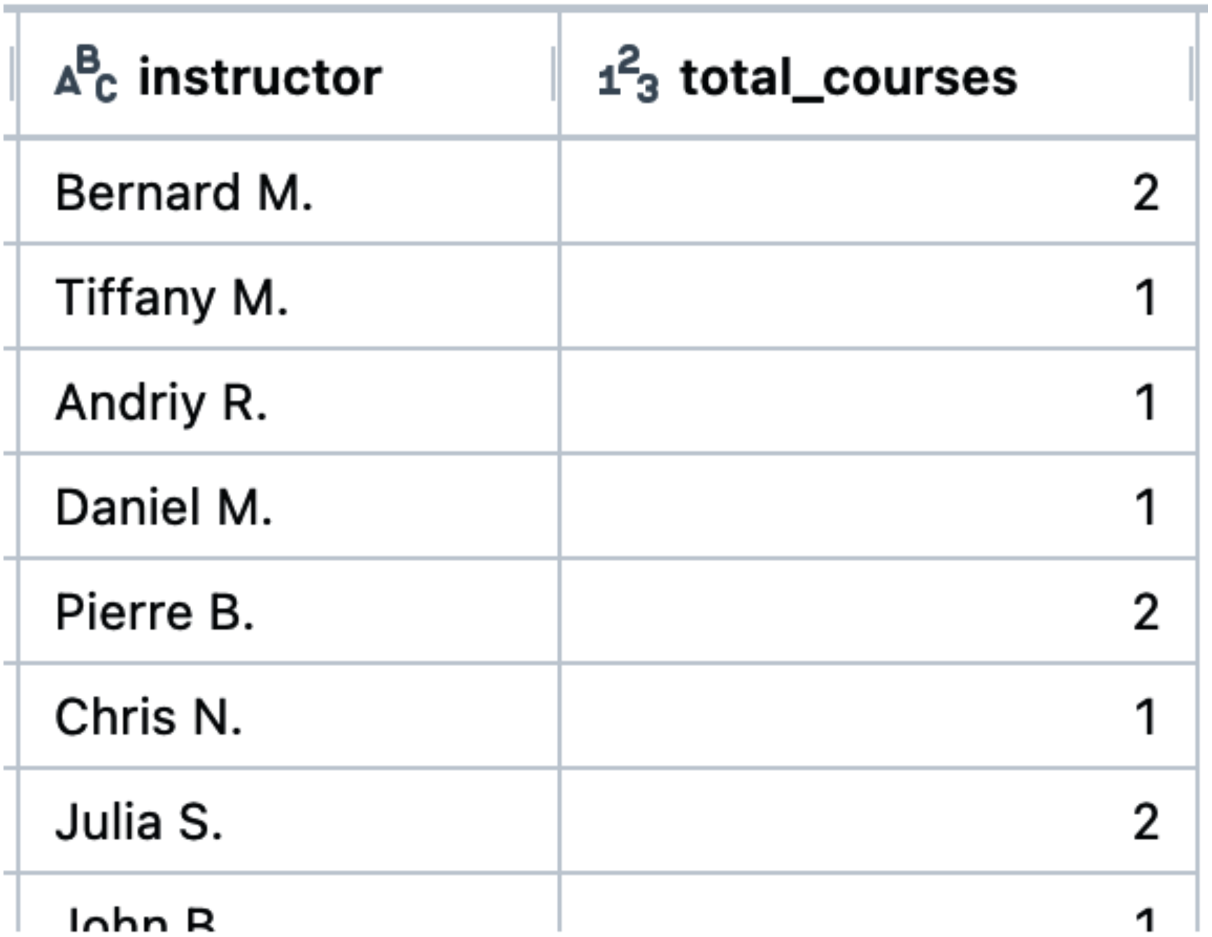
```
%sql
SELECT * FROM instructor_counts_py
```

Alternatively, you can use the PySpark DataFrame API to query the table. This can be achieved using the `spark.read` method:

```
instructor_counts_df = spark.read.table("instructor_counts_py")

display(instructor_counts_df)
```

In this code snippet, the `spark.read` method is used to create a static DataFrame against our table. Then, the display function is invoked to show the queried data, as shown in Figure 5-12.



| instructor | total_courses |
|---|---|
| Bernard M. | 2 |
| Tiffany M. | 1 |
| Andriy R. | 1 |
| Daniel M. | 1 |
| Pierre B. | 2 |
| Chris N. | 1 |
| Julia S. | 2 |
| John B | 1 |

Figure 5-12. The result of querying the `instructor_count_py` table

In conclusion, Spark Structured Streaming provides a powerful and flexible solution for handling streaming data processing tasks. By using either Spark SQL or PySpark DataFrame APIs, you can perform a variety of data manipulations on streaming data sources, including Delta Lake. This enables you to build end-to-end reliable data pipelines for a wide range of use cases, from real-time analytics to incremental data ingestion, as you are about to see.

# Incremental Data Ingestion

Data ingestion is a crucial step in data engineering pipelines, particularly when dealing with files stored in cloud storage. In this section, we will explore the different techniques of incrementally loading data from files into Delta Lake. Our focus will be on two primary methods: the `COPY INTO` command and Auto Loader.

## Introducing Data Ingestion

Data ingestion, as used in this book, refers to the process of loading data from files into Delta Lake tables. One of the significant challenges in data ingestion is the need to avoid reprocessing the same files multiple times. In a traditional data pipeline, each time the pipeline is run, it would reprocess all the files, including those that have already been ingested previously. This approach can be computationally expensive, time-consuming, and can lead to additional deduplication work, especially when dealing with large datasets, and this is where incremental data ingestion comes into play.

Incremental data ingestion involves loading only files newly received since the last data ingestion cycle. This approach ensures that data pipelines are optimized by avoiding the reprocessing of previously ingested files, thereby reducing the processing time and resources required. Databricks offers two efficient mechanisms for the incremental processing of newly arrived files in a storage location: the `COPY INTO` SQL command and Auto Loader. Let us examine these methods in detail and learn how to implement them effectively.

## COPY INTO Command

The `COPY INTO` command is a SQL statement that facilitates the loading of data from a specified file location into a Delta table. This command operates in an idempotent and incremental manner, meaning that each execution will only process new files from the source location, while previously ingested files are ignored.
The syntax for the `COPY INTO` command is straightforward and is structured as follows:

```sql
COPY INTO my_table
FROM '/path/to/files'
FILEFORMAT = <format>
FORMAT_OPTIONS (<format options>)
```

```
5 COPY_OPTIONS (<copy options>)
```

The command structure specifies the target table (line 1), the source file location (line 2), the format of the source files such as CSV or Parquet (line 3), any pertinent file options (line 4), and additional options to control the ingestion operation (line 5).

For instance:

```
COPY INTO my_table
FROM '/path/to/files'
FILEFORMAT = CSV
FORMAT_OPTIONS ('delimiter' = '|',
                'header' = 'true')
COPY_OPTIONS ('mergeSchema' = 'true')
```

In this example, the command is configured to ingest data into a Delta Lake table, named `my_table`, from a given source location. This location contains CSV files characterized by having headers and a specific delimiter, |. Furthermore, the `COPY_OPTIONS` parameter is leveraged to facilitate schema evolution in response to modifications in the structure of the incoming data.

## Auto Loader

The second method for loading data incrementally from files in Databricks is Auto Loader. It leverages Structured Streaming in Spark to efficiently process new data files as they become available in a storage location. Notably, Auto Loader offers scalability by allowing for handling billions of files and supporting real-time ingestion rates of millions of files per hour.

Built upon Spark’s Structured Streaming framework, Auto Loader employs checkpointing to track the ingestion process and store metadata information about the discovered files. This ensures that data files are processed exactly once by Auto Loader. Moreover, in the event of a failure, Auto Loader seamlessly resumes processing from the point of interruption.

### Implementation

As an integral part of Spark’s Structured Streaming, you can work with Auto Loader by using the `readStream` and `writeStream` methods:

```
spark.readStream
          .format("cloudFiles")
          .option("cloudFiles.format", <source_format>)
          .load(’/path/to/files’)
      .writeStream
          .option("checkpointLocation", <checkpoint_directory>)
          .table(<table_name>)
```

Auto Loader introduces a specific format of DataStreamReader named `cloudFiles`. The `cloudFiles.format` option is employed to specify the format of the source files. Then, the load function is used to indicate the location of the source files, where Auto Loader detects and queues new arrivals for ingestion. Subsequently, data is written into a target table using the DataStreamWriter, with the `checkpointLocation` parameter indicating where checkpointing information should be stored.

### Schema management

Auto Loader offers a convenient feature that enables automatic schema detection for loaded data, allowing you to create tables without explicitly defining the data schema. Moreover, if new columns are added, the table schema can evolve accordingly. However, to avoid inference costs during each stream startup, the inferred schema can be stored for subsequent use. This is achieved by specifying a location where Auto Loader can store the schema using the `cloudFiles.schemaLocation` option.

Note that the schema inference behavior of Auto Loader varies depending on the file format. For formats with typed schemas, such as Parquet, Auto Loader extracts the predefined schemas from the files. On the other hand, for formats that don't encode data types, like JSON and CSV, Auto Loader infers all columns as strings by default. To enable inferring column data types from such sources, you can set the option `cloudFiles.inferColumnTypes` to true:

```
spark.readStream
          .format("cloudFiles")
          .option("cloudFiles.format", <source_format>)
          .option("cloudFiles.inferColumnTypes", "true")
          .option("cloudFiles.schemaLocation", <schema_directory>)
          .load('/path/to/files')
     .writeStream
          .option("checkpointLocation", <checkpoint_directory>)
          .option("mergeSchema", "true")
          .table(<table_name>)
```

It's worth mentioning that the designated schema location can be identical to the checkpoint location for simplicity and convenience; they will not conflict.

## Comparison of Ingestion Mechanisms

When deciding between the `COPY INTO` command and Auto Loader for your data ingestion tasks, it's important to consider two key factors, which are summarized in Table 5-3.

| | `COPY INTO` | **Auto Loader** |
|---|---|---|
| **File volume** | Thousands of files | Millions of files |

| | COPY INTO | **Auto Loader** |
|---|---|---|
| **Efficiency** | Less efficient at scale | Efficient at scale |

Table 5-3. Comparison of the incremental data ingestion mechanisms

File volume

The `COPY INTO` command is ideal for scenarios where the volume of incoming files is relatively small, typically on the order of thousands. It offers simplicity and straightforward execution, making it well-suited for smaller-scale data ingestion tasks. On the other hand, Auto Loader is suited for scenarios where the volume of incoming files is on the order of millions or more over time.

Efficiency

Auto Loader has the capability to split processing into multiple batches, thereby enabling faster and more efficient data ingestion compared to the `COPY INTO` command. This attribute makes Auto Loader an ideal choice for environments characterized by high data velocity and volume.

As a general best practice, Databricks recommends using Auto Loader when ingesting data from cloud object storage.

## Auto Loader in Action

Let's walk through the practical implementation of Auto Loader for incremental data ingestion from files. We will continue using our online school dataset, consisting of three tables: `students`, `enrollments`, and `courses`.

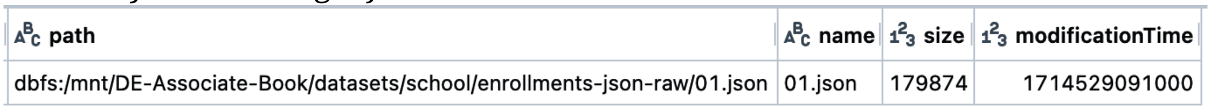
In this demonstration, we will use a new Python notebook titled "5.2 - Auto Loader." We begin by running the "School-Setup" helper notebook to prepare our environment:

```
%run ../Includes/School-Setup
```

In this scenario, we will leverage Auto Loader to incrementally ingest student enrollment data from JSON files into a target Delta table. Before setting up our Auto Loader stream, let's inspect our source directory:

```
files = dbutils.fs.ls(f"{dataset_school}/enrollments-json-raw")
display(files)
```

Figure 5-13 displays the contents of the source directory, showing that it currently hosts a single JSON file.



| path | name | size | modificationTime |
|---|---|---|---|
| dbfs:/mnt/DE-Associate-Book/datasets/school/enrollments-json-raw/01.json | 01.json | 179874 | 1714529091000 |

Figure 5-13. The content of the source directory

Now, we'll set up Auto Loader to efficiently handle the ingestion of this file and any new files arriving in the directory.

### Setting up Auto Loader

Remember, Auto Loader uses the `readStream` and `writeStream` methods from Spark's Structured Streaming API. Here's an example of how to set up Auto Loader for our use case:

```
(spark.readStream
             .format("cloudFiles")
             .option("cloudFiles.format", "json")
             .option("cloudFiles.inferColumnTypes","true")
             .option("cloudFiles.schemaLocation",
                       "dbfs:/mnt/DEA-Book/checkpoints/enrollments")
             .load(f"{dataset_school}/enrollments-json-raw")
      .writeStream
             .option("checkpointLocation",
                       "dbfs:/mnt/DEA-Book/checkpoints/enrollments")
             .table("enrollments_updates")
)
```

In this configuration, the `cloudFiles` format represents the Auto Loader stream, with three additional options:

*`cloudFiles.format`*
: Specifies the format of the data files being ingested, in this case, JSON.

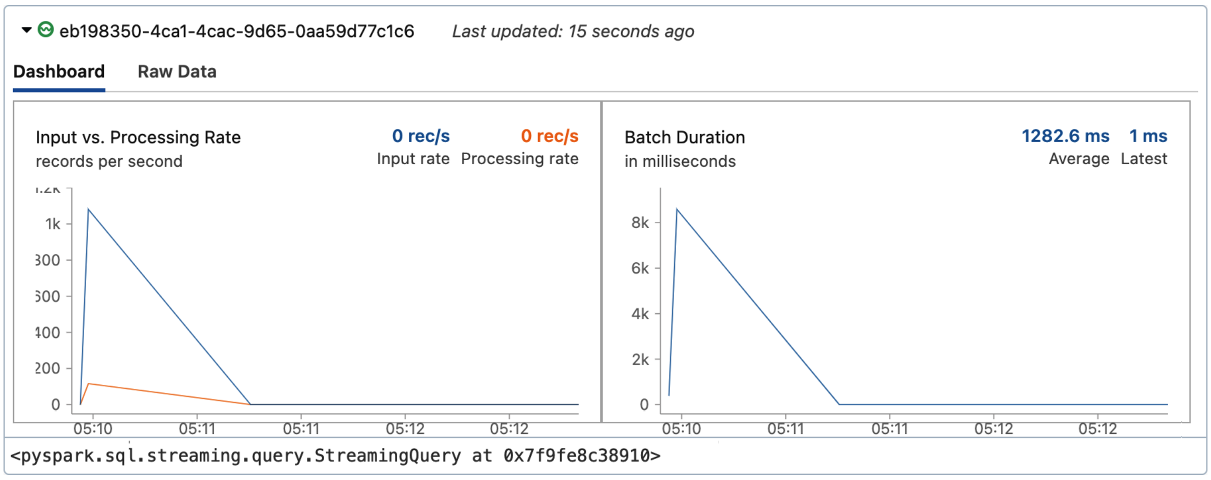
*`cloudFiles.inferColumnTypes`*
: Enables Auto Loader to automatically determine the data types of the columns.

*`cloudFiles.schemaLocation`*
: Sets the directory where Auto Loader can store the inferred schema information.

Subsequently, we use the load method to define the location of our data source files.

Following that, we immediately chain a `writeStream` method to write the ingested data into a target table called `enrollments_updates`. Furthermore, we provide a location for storing checkpoint information, enabling Auto Loader to track the ingestion process. It's worth noting that both schema and checkpoint information are stored within the same directory.

Upon executing the previous command, a streaming query is initiated, as illustrated in Figure 5-14. This query remains active, continuously processing new data as it arrives in the data source directory.



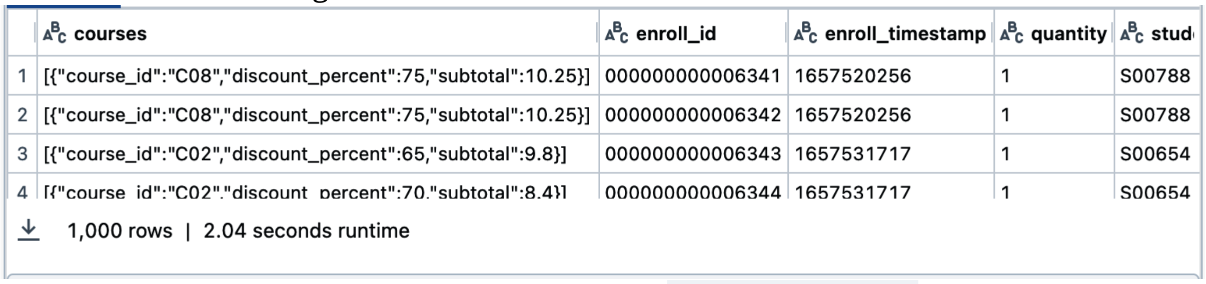
```
<pyspark.sql.streaming.query.StreamingQuery at 0x7f9fe8c38910>
```

Figure 5-14. Streaming write operation by Auto Loader

To confirm the successful data ingestion, we can review the contents of the updated table by executing a standard `SELECT` statement:

```
%sql
SELECT * FROM enrollments_updates
```

Figure 5-15 displays the result of querying our target table after the initial ingestion. At this point, the table contains 1,000 records, confirming that our stream is correctly configured and that data is being successfully processed and stored in the target table.



| | courses | enroll_id | enroll_timestamp | quantity | stud |
|---|---|---|---|---|---|
| 1 | [{"course_id":"C08","discount_percent":75,"subtotal":10.25}] | 000000000006341 | 1657520256 | 1 | S00788 |
| 2 | [{"course_id":"C08","discount_percent":75,"subtotal":10.25}] | 000000000006342 | 1657520256 | 1 | S00788 |
| 3 | [{"course_id":"C02","discount_percent":65,"subtotal":9.8}] | 000000000006343 | 1657531717 | 1 | S00654 |
| 4 | [{"course_id":"C02","discount_percent":70,"subtotal":8.4}] | 000000000006344 | 1657531717 | 1 | S00654 |

1,000 rows | 2.04 seconds runtime

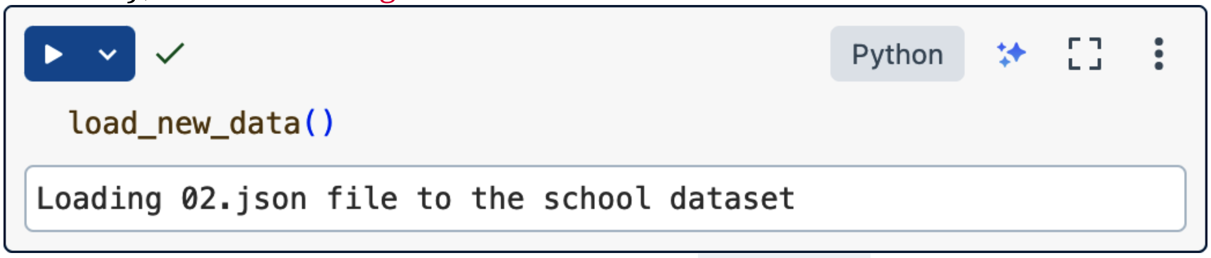
Figure 5-15. The result of querying the `enrollments_updates` table

## Observing Auto Loader

As part of this demonstration, we can simulate an external system adding new data files to our source directory. This is achieved by the `load_new_data` helper function, which is provided with our online school dataset. Each execution of this function mimics the external system adding a single file of 1,000 records:

```
load_new_data()
```

After running the command, a new file is successfully copied to our source directory, as shown in Figure 5-16.



```
load_new_data()
```

```
Loading 02.json file to the school dataset
```

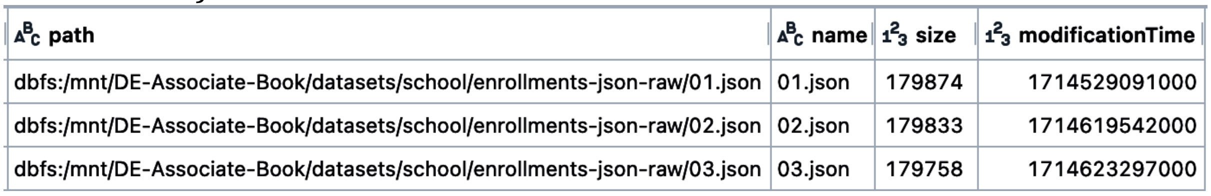
Figure 5-16. The output of executing the `load_new_data` function

To further increase the volume of data for our demonstration, let's run the previous command a second time for adding another file.

With two new files added, we can re-examine the contents of our source directory to confirm their presence:

```
files = dbutils.fs.ls(f"{dataset_school}/enrollments-json-raw")
display(files)
```

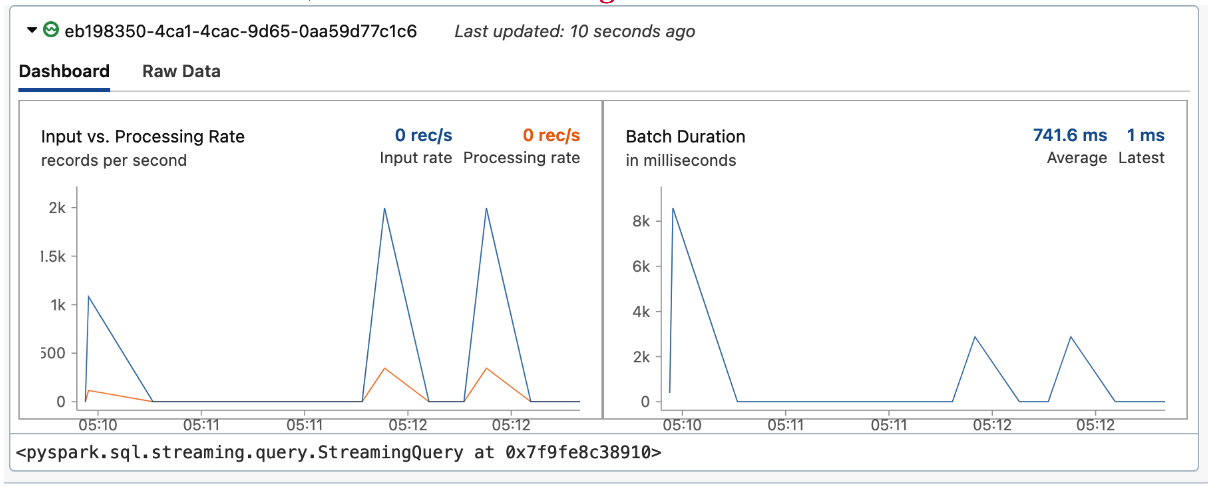
Figure 5-17 displays the updated contents of the source directory, confirming the addition of two new files. Remember, our Auto Loader stream is still active, continuously scanning the directory for new files and processing any that are detected. With this set up, the new files will be processed automatically.



| path | name | size | modificationTime |
|---|---|---|---|
| dbfs:/mnt/DE-Associate-Book/datasets/school/enrollments-json-raw/01.json | 01.json | 179874 | 1714529091000 |
| dbfs:/mnt/DE-Associate-Book/datasets/school/enrollments-json-raw/02.json | 02.json | 179833 | 1714619542000 |
| dbfs:/mnt/DE-Associate-Book/datasets/school/enrollments-json-raw/03.json | 03.json | 179758 | 1714623297000 |

Figure 5-17. The content of the source directory after landing a new data file

Returning to our Auto Loader stream above, you can observe its current activity through the provided dashboard. It indeed indicates the reception of new batches of data, as illustrated in Figure 5-18.



```
<pyspark.sql.streaming.query.StreamingQuery at 0x7f9fe8c38910>
```

Figure 5-18. Auto Loader stream processing after landing two new data files

To confirm that the new data has been successfully processed and ingested into our target table, we can check the number of records in the table:

```
%sql
SELECT count(*) FROM enrollments_updates
```

This command reveals that our `enrollments_updates` table now has a total of 3,000 records, confirming the insertion of data from the new files. This highlights Auto Loader's capability to detect and process new files as soon as they appear in the source directory, demonstrating its efficiency and reliability for near-real-time data ingestion. Of course, you can also execute

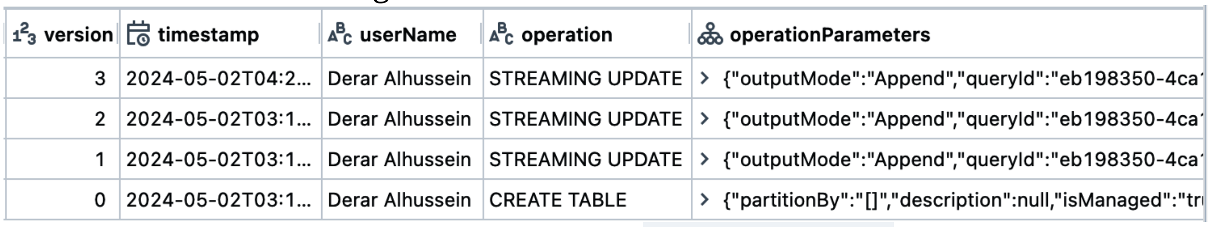
Auto Loader in incremental batch mode by using the `availableNow` trigger option.

### Exploring table history

After successfully updating our Delta Lake table using Auto Loader, it's valuable to review the history of changes made to the table during this process. To achieve this, let's run the `DESCRIBE HISTORY` command on the `enrollments_updates` table:

```
%sql
DESCRIBE HISTORY enrollments_updates
```

Figure 5-19 displays our table history, revealing three new table versions, each corresponding to an update triggered by the Auto Loader stream. It's evident that each of these entries aligns with the arrival of one of our data files at the source. Note in particular that `writeSteam` registers the operation as a streaming update rather than a normal write operation (see Chapter 2 for more on the Dela Lake transaction log).



| version | timestamp | userName | operation | operationParameters |
|---|---|---|---|---|
| 3 | 2024-05-02T04:2... | Derar Alhussein | STREAMING UPDATE | > {"outputMode":"Append","queryId":"eb198350-4ca |
| 2 | 2024-05-02T03:1... | Derar Alhussein | STREAMING UPDATE | > {"outputMode":"Append","queryId":"eb198350-4ca |
| 1 | 2024-05-02T03:1... | Derar Alhussein | STREAMING UPDATE | > {"outputMode":"Append","queryId":"eb198350-4ca |
| 0 | 2024-05-02T03:1... | Derar Alhussein | CREATE TABLE | > {"partitionBy":"[]","description":null,"isManaged":"tr |

Figure 5-19. The history log of the `enrollments_updates` table

### Cleaning up

At the end of this demonstration, we can tidy up by performing two cleanup actions: dropping the table and removing the checkpoint directory. However, let's first revisit our Auto Loader query and stop the active streaming job.

With the streaming job interrupted, we can proceed to drop the `enrollments_updates` table:

```
%sql
DROP TABLE enrollments_updates
```

Finally, we remove the checkpoint location associated with our Auto Loader stream by running the `dbutils.fs.rm` function:

```
dbutils.fs.rm("dbfs:/mnt/DEA-Book/checkpoints/enrollments", True)
```

In summary, Auto Loader has proven to be a powerful tool for automating the data ingestion process, allowing for efficient and scalable data loading. Its ability to handle high volumes of data makes it an essential component of many modern data pipelines. In the next section, we'll explore how to take Auto Loader to the next level by using it in a medallion architecture, enabling even more complex and scalable data processing workflows.

## Medallion Architecture

A medallion architecture is a robust approach for efficiently processing data through multiple stages of transformation. In this section, we will delve into the fundamental concepts and benefits of this architecture. Following this, we will explore a step-by-step guide on implementing a medallion architecture on the Databricks platform.

## Introducing Medallion Architecture

A medallion architecture, also referred to as multi-hop architecture, is a data design pattern that logically organizes data in a multi-layered approach. Its primary objective is to gradually enhance both the structure and the quality of data as it progresses through successive processing layers.

### The layered approach

The medallion architecture is structured into three principal layers, each serving a distinct purpose in the data refinement process. These layers are symbolically termed bronze, silver, and gold, indicating their ascending order of quality and value, as illustrated in Figure 5-20.

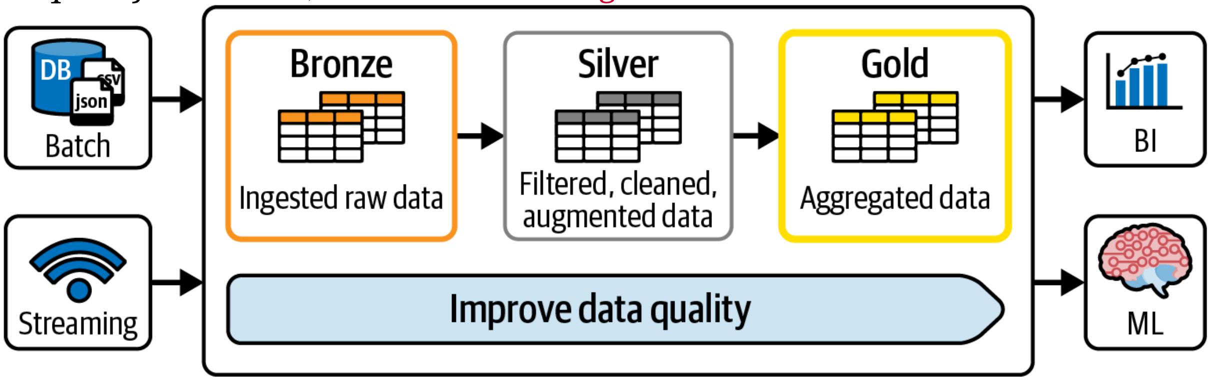


Figure 5-20. Medallion architecture

Let's dive deeper to gain a comprehensive understanding of each of these layers.

#### *Bronze layer*

The foundation of the medallion architecture starts at the bronze layer, which is the initial stage of data ingestion. At this layer, data is ingested from external systems and stored in its rawest form. This raw data is retained in tables known as bronze tables, which serve as repositories for unprocessed data. The data stored in these tables is exactly as it was received from the source systems, without any transformations or modifications. This approach ensures that the original data is preserved, preventing data loss and enabling easy auditing and traceability.

The data sources that feed into the bronze layer are diverse and varied. They can range from structured data files to operational databases. Moreover, the bronze tables are also common destinations for streaming data from platforms like Kafka, enabling real-time data ingestion and processing.

So, the bronze layer’s primary function is to ensure that all data is captured and stored, regardless of its source or quality. This provides a comprehensive snapshot of information in its original form, serving as a single source of truth for your data projects.

#### *Silver layer*

As data moves up to the silver layer, it goes through significant processing to improve its quality and utility for further analysis. This middle layer focuses on data cleansing, normalization, and validation. Incorrect or irrelevant data points are filtered out, and inconsistencies are resolved to ensure data reliability. Moreover, this stage often involves enriching the data by joining fields from various datasets, thereby providing a more integrated and coherent view. For instance, data from different departmental databases might be consolidated to provide a comprehensive view of organizational operations.

So, the enhancements made at the silver layer are designed to prepare the data for various analytical tasks that require a higher degree of data integrity and accuracy.

#### *Gold layer*

The final layer is the gold tables, where data reaches its most business-ready form. This layer is characterized by its role in facilitating high-level business analytics and intelligence. Data at this stage is often aggregated and summarized to support specific business needs, such as performance metrics, financial summaries, and customer insights.

So, the transformations at the gold layer make the data ready for reporting, dashboarding, and advanced analytics in machine learning and AI.

### Benefits of Medallion Architectures

The medallion architecture offers several advantages that can be summarized by the following key points:

*Simplicity*

The architecture represents a simplified data model that is intuitive and easy to understand and implement. By organizing data into distinct layers, each serving a specific purpose, the complexity of data management and maintenance is significantly reduced.

*Incremental ETL*

This architecture enables incrementally transforming and loading data as it arrives. This facilitates integrating new data and propagating it through each layer of the architecture.

*Hybrid workloads*

The architecture offers the flexibility to combine both streaming and batch processing within a unified pipeline. Each stage can be configured to operate either as a batch or a streaming job, depending on the nature of the data and the desired processing latency.

*Table reconstruction*

Another major benefit of this architecture is the ability to regenerate downstream tables from raw data at any time. This capability is particularly valuable in scenarios where data quality issues are detected during post-processing and must be solved at the source.

## Building Medallion Architectures

In this section, we will walk through a step-by-step process to implement a complete medallion architecture in Databricks. As a practical example, we will demonstrate how to manage our school enrollments using this approach. So, we will continue using our dataset, consisting of three tables: `students`, `enrollments`, and `courses`.

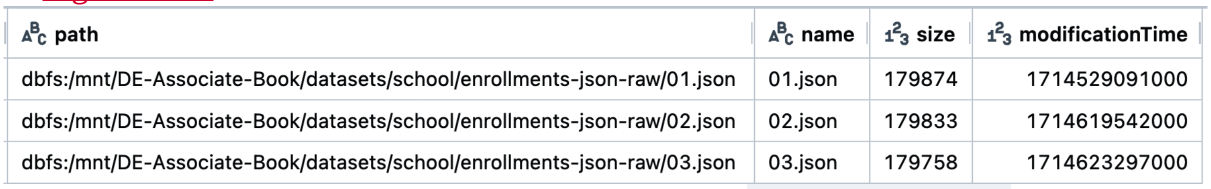
In this exercise, we will use a new Python notebook titled “5.3 - Medallion Architecture.” We begin by running the “School-Setup” helper notebook to prepare our environment:

```
%run ../Includes/School-Setup
```

Let’s start by revisiting the contents of our source directory:

```
files = dbutils.fs.ls(f"{dataset_school}/enrollments-json-raw")
display(files)
```

At present, there are three JSON files within the directory, as illustrated in Figure 5-21.



| path | name | size | modificationTime |
|---|---|---|---|
| dbfs:/mnt/DE-Associate-Book/datasets/school/enrollments-json-raw/01.json | 01.json | 179874 | 1714529091000 |
| dbfs:/mnt/DE-Associate-Book/datasets/school/enrollments-json-raw/02.json | 02.json | 179833 | 1714619542000 |
| dbfs:/mnt/DE-Associate-Book/datasets/school/enrollments-json-raw/03.json | 03.json | 179758 | 1714623297000 |

Figure 5-21. The content of the source directory `enrollments-json-raw`

These files represent the raw material of our data pipeline, awaiting ingestion into the bronze layer tables.

## Establishing the bronze layer

Our journey of implementing a medallion architecture begins in the bronze layer, which is the foundational layer for data ingestion. It serves as the initial repository that captures all incoming data in its rawest form, before any transformation or cleansing occurs.

### *Configuring Auto Loader*

The first step in the bronze layer typically involves configuring Auto Loader against the source directory. Here, we configure our Auto Loader stream to process the input files and load the data into a Delta Lake table:

```
import pyspark.sql.functions as F

(spark.readStream
             .format("cloudFiles")
             .option("cloudFiles.format", "json")
             .option("cloudFiles.inferColumnTypes","true")
             .option("cloudFiles.schemaLocation",
                       f"{checkpoint_path}/enrollments_bronze")
             .load(f"{dataset_school}/enrollments-json-raw")
             .select("*",
                       F.current_timestamp().alias("arrival_time"),
                       F.input_file_name().alias("source_file"))
      .writeStream
             .format("delta")
             .option("checkpointLocation",
                       f"{checkpoint_path}/enrollments_bronze")
             .outputMode("append")
             .table("enrollments_bronze")
)
```

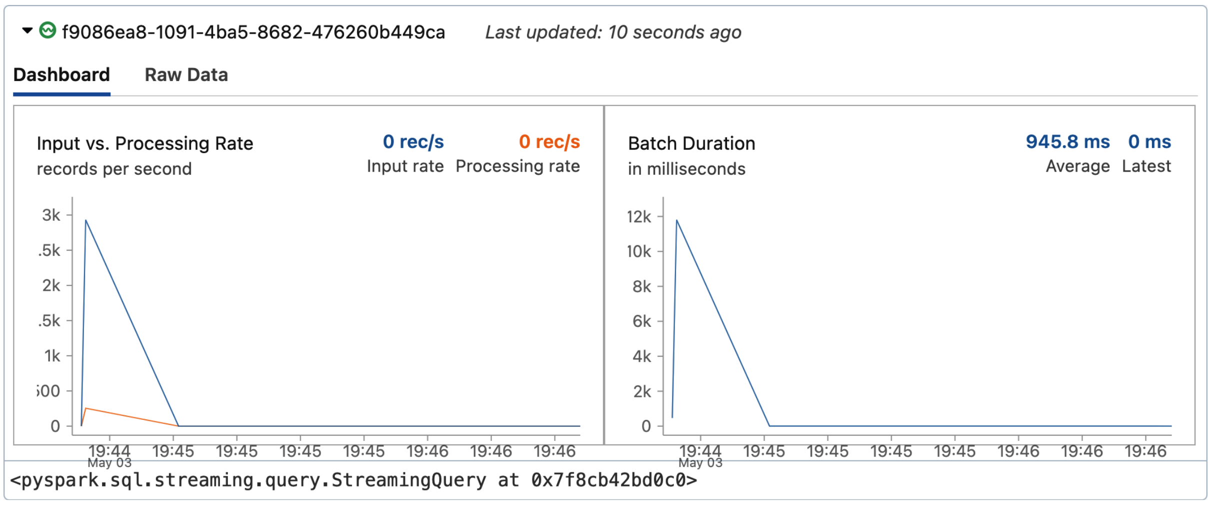
In this segment, we start by initiating a streaming read operation from our JSON source files. The reader is set to infer the columns' data types automatically, ensuring that they are correctly identified without explicit declaration. The data is then combined with two supplementary pieces of metadata available through Auto Loader:

*`arrival_time`*
: Timestamp of when the data is ingested, which is valuable for tracking and auditing purposes.

*`source_file`*
: The name of the file from which the data is sourced, aiding in data lineage and troubleshooting.

After the data is read and supplemented with this metadata, it is streamed directly into a Delta table named `enrollments_bronze`.

Upon activating this Auto Loader stream, we can observe that a new batch of data has been detected and processed, as illustrated in Figure 5-22.



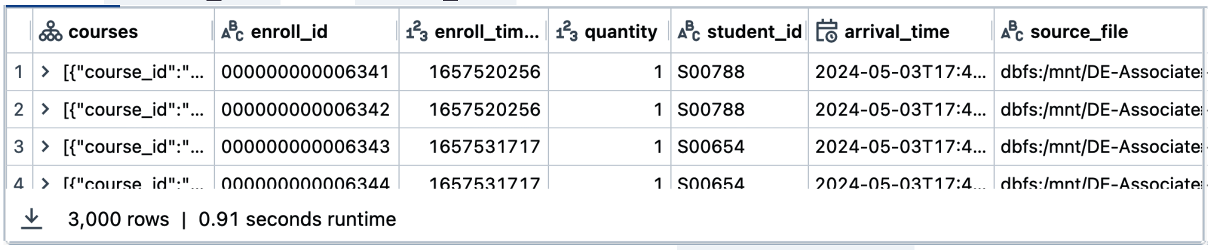
```
<pyspark.sql.streaming.query.StreamingQuery at 0x7f8cb42bd0c0>
```

Figure 5-22. Streaming write operation by Auto Loader

To inspect the raw data that has been captured, we can simply query the `enrollments_bronze` table:

```
%sql
SELECT * FROM enrollments_bronze
```

Figure 5-23 displays the result of this query, confirming the successful ingestion of the data along with the added metadata fields: `arrival_time` and `source_file`.



| | courses | enroll_id | enroll_tim... | quantity | student_id | arrival_time | source_file |
|---|---|---|---|---|---|---|---|
| 1 | > [{"course_id":"... | 000000000006341 | 1657520256 | 1 | S00788 | 2024-05-03T17:4... | dbfs:/mnt/DE-Associate |
| 2 | > [{"course_id":"... | 000000000006342 | 1657520256 | 1 | S00788 | 2024-05-03T17:4... | dbfs:/mnt/DE-Associate |
| 3 | > [{"course_id":"... | 000000000006343 | 1657531717 | 1 | S00654 | 2024-05-03T17:4... | dbfs:/mnt/DE-Associate |
| 4 | > [{"course_id":"... | 000000000006344 | 1657531717 | 1 | S00654 | 2024-05-03T17:4... | dbfs:/mnt/DE-Associate |

3,000 rows | 0.91 seconds runtime

Figure 5-23. The result of querying the `enrollments_bronze` table

Next, we can verify the volume of data that has been written into the bronze layer:

```
%sql
SELECT count(1) FROM enrollments_bronze
```

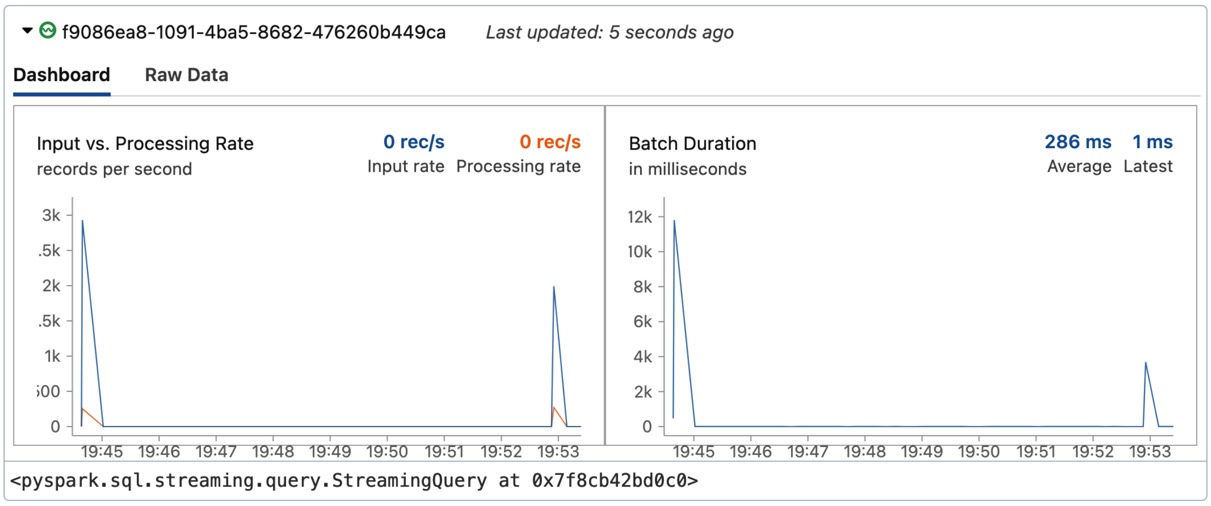
This command reveals that 3,000 records have been persisted, which corresponds to our three source files, each containing 1,000 records. This confirms that our ingestion process is correctly configured and functioning as expected.

To demonstrate the stream processing capabilities of our data pipeline, let's simulate the arrival of new data in the source directory using our `load_new_data` function:

```
load_new_data()

Output: Loading 04.json file to the school dataset
```

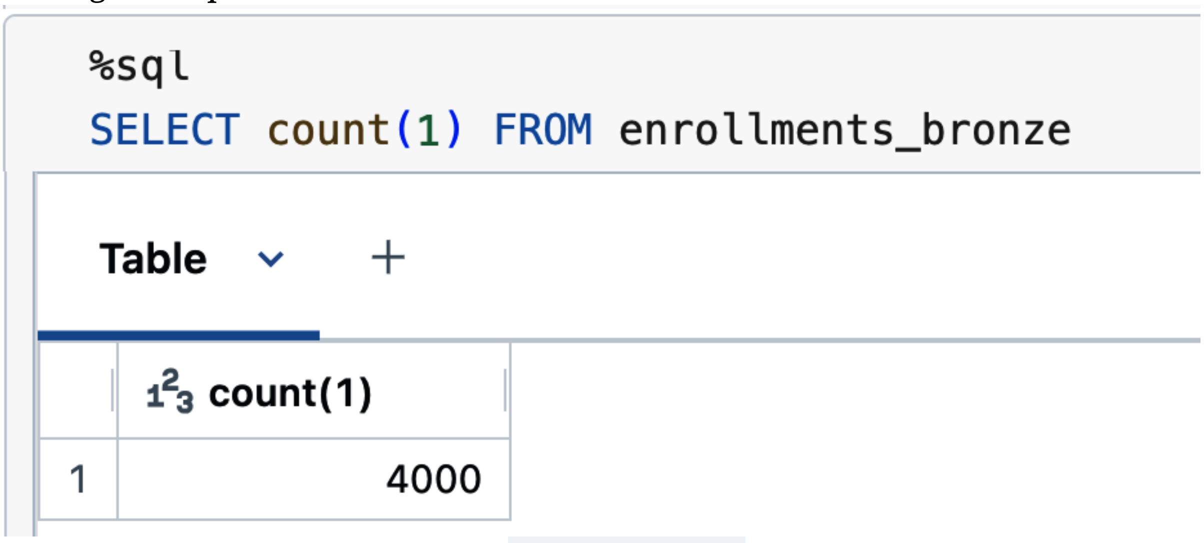
Returning to our previous active stream, we observe that the new data is immediately detected and processed by the streaming query, as illustrated in Figure 5-24.



```
<pyspark.sql.streaming.query.StreamingQuery at 0x7f8cb42bd0c0>
```

Figure 5-24. Auto Loader stream processing after landing the new data file

By re-querying the number of records in the bronze table, we can verify that the new data has been successfully ingested. As shown in Figure 5-25, the table now contains 4,000 records, reflecting an increase of 1,000 records since our last ingestion process.



```
%sql
SELECT count(1) FROM enrollments_bronze
```

| | count(1) |
|---|---|
| 1 | 4000 |

Figure 5-25. The number of records in the `enrollments_bronze` table after loading the new input file

### *Creating a static lookup table*

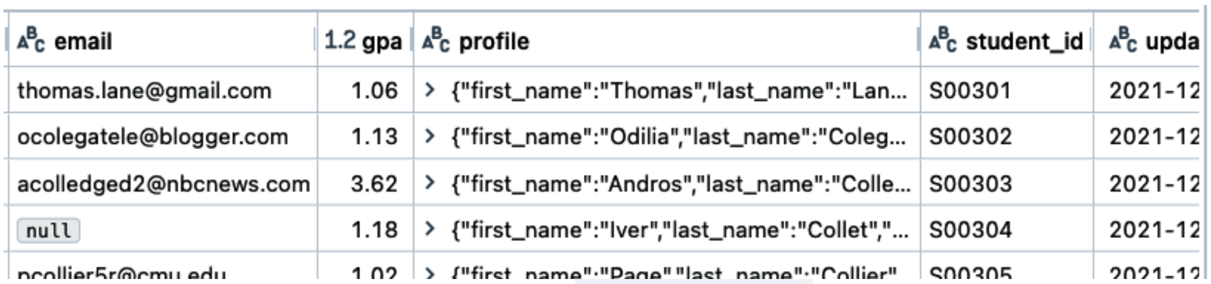
In preparation for data processing within the subsequent layers, we may need to integrate additional data sources that can enrich our primary datasets. In our case, we require a static lookup table of student information. This table will be used in the silver layer to join with the enrollment data in order to add more depth and context to our analysis. To create our static lookup, we use the `spark.read` method to construct a static DataFrame from the students' JSON files:

```
students_lookup_df = (spark.read
                           .format("json")
                           .load(f"{dataset_school}/students-json"))
```

Before proceeding further, let's examine the structure and contents of the newly created lookup DataFrame:

```
display(students_lookup_df)
```

The results, visualized in Figure 5-26, illustrate that the `students` lookup DataFrame consists of several columns such as student ID, email, and profile information.



| email | 1.2 gpa | profile | student_id | upda |
|---|---|---|---|---|
| thomas.lane@gmail.com | 1.06 | > {"first_name":"Thomas","last_name":"Lan... | S00301 | 2021-12 |
| ocolegatele@blogger.com | 1.13 | > {"first_name":"Odilia","last_name":"Coleg... | S00302 | 2021-12 |
| acolledged2@nbcnews.com | 3.62 | > {"first_name":"Andros","last_name":"Colle... | S00303 | 2021-12 |
| null | 1.18 | > {"first_name":"Iver","last_name":"Collet","... | S00304 | 2021-12 |
| pcollier5r@cmu.edu | 1.02 | > {"first_name":"Page","last_name":"Collier" | S00305 | 2021-12 |

Figure 5-26. Displaying the `students_lookup_df` DataFrame

With our bronze layer established, we can now progress to the next phase of our data processing pipeline—the silver layer.

### Transitioning to the silver layer

In the silver layer, our focus shifts to refining and enhancing the data acquired from the bronze layer. At this stage, we refine the raw data by adding contextual information, formatting values, and performing data quality checks. Our objective is to ensure that the data is clean, structured, and optimized for downstream processing and analysis.

In the following code snippet, we initiate a streaming read operation on the `enrollments_bronze` table, and then we apply a series of transformations to enrich and refine the data:

```
enrollments_enriched_df = (spark.readStream
      .table("enrollments_bronze")
      .where("quantity > 0")
      .withColumn("formatted_timestamp",
                  F.from_unixtime("enroll_timestamp",
                                  "yyyy-MM-dd HH:mm:ss").cast("timestamp") )
      .join(students_lookup_df, "student_id")
      .select("enroll_id", "quantity", "student_id", "email",
                "formatted_timestamp", "courses")
)
```

The transformations applied in this step include the following:

*Data cleansing*
: We exclude any enrollments with no items (quantity > 0), ensuring that only valid records are processed further.

*Timestamp formatting*
: We parse the enrollment timestamp from the Unix time format into a human-readable format using the `from_unixtime` function to facilitate easier understanding and interpretation.

*Data enrichment*

We enrich the enrollment data by joining it with the student information from our static lookup DataFrame `students_lookup_df`. This adds the students' email addresses to the enrollment records.

*Column selection*

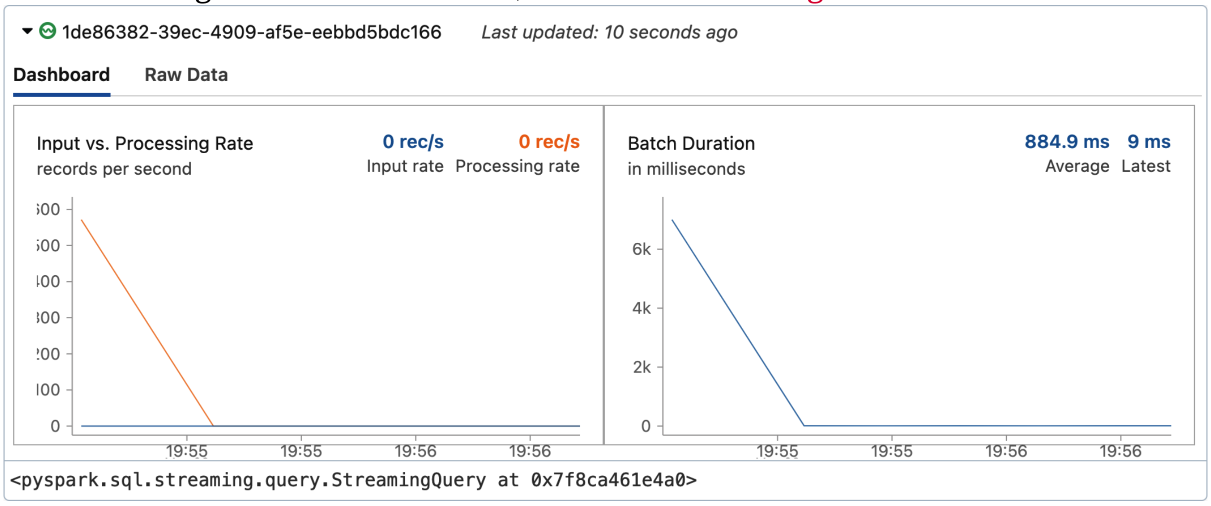
Finally, we select specific columns of interest for further analysis, including enrollment ID, quantity, student ID, email, formatted timestamp, and course information.

These transformations are executed using the PySpark API. However, it's worth noting that similar operations can also be achieved using Spark SQL. By registering a streaming temporary view against the bronze table, we can leverage SQL queries to perform the same transformations, just like we did earlier in this chapter.

Subsequently, we proceed to persist this processed streaming data into a dedicated silver table. We accomplish this by performing a stream write operation on the `enrollments_enriched_df` DataFrame:

```
(enrollments_enriched_df.writeStream
                            .format("delta")
                            .option("checkpointLocation",
                                     f"{checkpoint_path}/enrollments_silver")
                            .outputMode("append")
                            .table("enrollments_silver"))
```

This code snippet sets up a continuous streaming write into the `enrollments_silver` table. By specifying the output mode as `"append"`, new records will be added to the table as they are processed, ensuring that the table is incrementally populated with the latest data from the bronze layer. Upon executing the previous command, our stream is activated, and data starts flowing into the silver table, as illustrated in Figure 5-27.



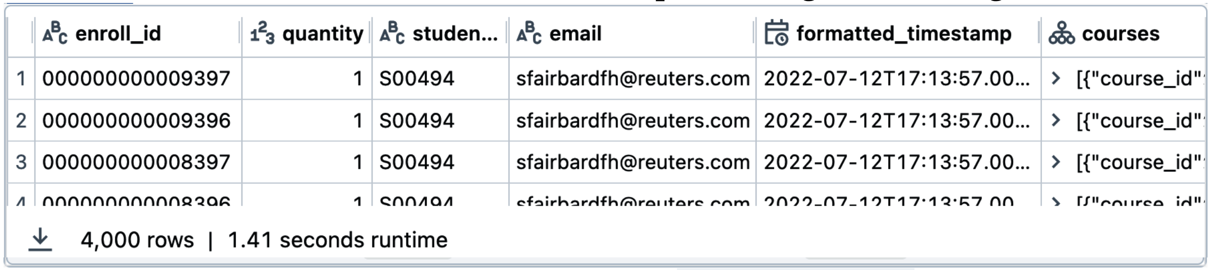
```
<pyspark.sql.streaming.query.StreamingQuery at 0x7f8ca461e4a0>
```

Figure 5-27. Stream processing in the silver layer

To verify the written data, let's query the `enrollments_silver` table:

```
%sql
SELECT * FROM enrollments_silver
```

Figure 5-28 displays the result of querying our silver table. The presence of all 4,000 records confirms the successful data processing and writing.



| | enroll_id | quantity | studen... | email | formatted_timestamp | courses |
|---|---|---|---|---|---|---|
| 1 | 000000000009397 | 1 | S00494 | sfairbardfh@reuters.com | 2022-07-12T17:13:57.00... | > [{"course_id" |
| 2 | 000000000009396 | 1 | S00494 | sfairbardfh@reuters.com | 2022-07-12T17:13:57.00... | > [{"course_id" |
| 3 | 000000000008397 | 1 | S00494 | sfairbardfh@reuters.com | 2022-07-12T17:13:57.00... | > [{"course_id" |
| 4 | 000000000008396 | 1 | S00494 | sfairbardfh@reuters.com | 2022-07-12T17:13:57.00 | > [{"course_id" |

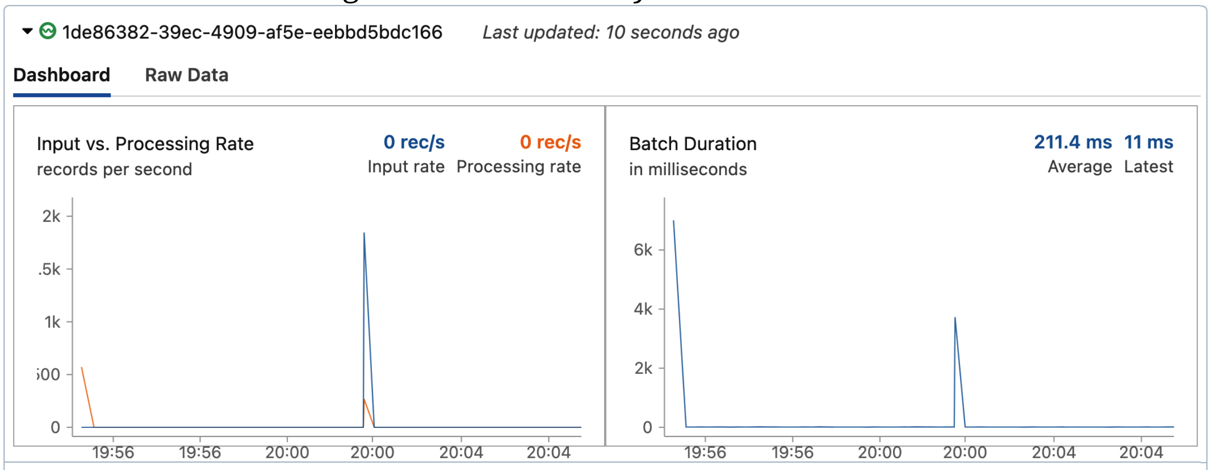
4,000 rows | 1.41 seconds runtime

Figure 5-28. The result of querying the `enrollments_silver` table

To further demonstrate the dynamic capabilities of our data pipeline, let's trigger the arrival of new data files in our source directory using the `load_new_data` function. We then monitor the propagation of this new data through the bronze layer and into the silver layer.

```
load_new_data()
Output: Loading 05.json file to the school dataset
```

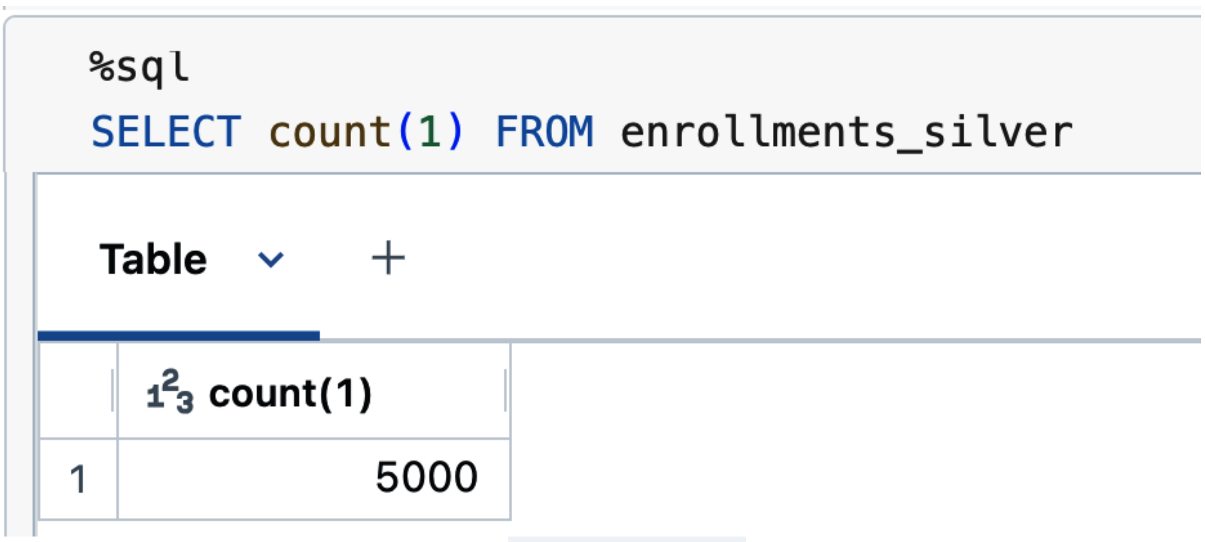
The new data now seamlessly propagates through the pipeline, starting from the active Auto Loader steam and continuing through to the silver layer. We can track the progress of processing using the dashboard associated with each stream. Figure 5-29 showcases the latest updates in the silver layer, confirming the successful handling of the new data by our stream.



```
<pyspark.sql.streaming.query.StreamingQuery at 0x7f8ca461e4a0>
```

Figure 5-29. Stream processing in the silver layer after landing the new data file

With the addition of 1,000 records from the latest file, the total count in the `enrollments_silver` table now stands at 5,000 records, as shown in Figure 5-30.



```
%sql
SELECT count(1) FROM enrollments_silver
```

| | count(1) |
|---|---|
| 1 | 5000 |

Figure 5-30. The number of records in the `enrollments_silver` table after loading the new input file

From here, we can now advance to the final phase of our medallion architecture—the gold layer.

### Advancing to the gold layer

In the gold layer, we concentrate on providing high-level aggregations and summaries. This layer is important for supporting business intelligence and analytics applications by presenting the data in its most refined form.

Our task here involves creating an aggregate table that summarizes the daily number of course enrollments per student. To accomplish this, we initiate a streaming read operation on the `enrollments_silver` table, and then we perform the necessary transformations to aggregate the data by student ID, email, and day:

```
enrollments_agg_df =(spark.readStream
             .table("enrollments_silver")
             .withColumn("day", F.date_trunc("DD", "formatted_timestamp"))
             .groupBy("student_id", "email", "day")
             .agg(F.sum("quantity").alias("courses_counts"))
             .select("student_id", "email", "day", "courses_counts")
)
```

In the previous code, the `date_trunc` function is used to truncate the timestamp to the day level, allowing us to group the data by day.
Once this aggregation logic is applied, we can proceed to persist the aggregated data into a dedicated gold table named `daily_student_courses`:

```
(enrollments_agg_df.writeStream
             .format("delta")
             .outputMode("complete")
             .option("checkpointLocation",
                      f"{checkpoint_path}/daily_student_courses")
             .trigger(availableNow=True)
             .table("daily_student_courses"))
```

In this configuration, we specify the output mode as `"complete"`, indicating that the entire aggregation result should be rewritten each time the logic runs.

**NOTE**

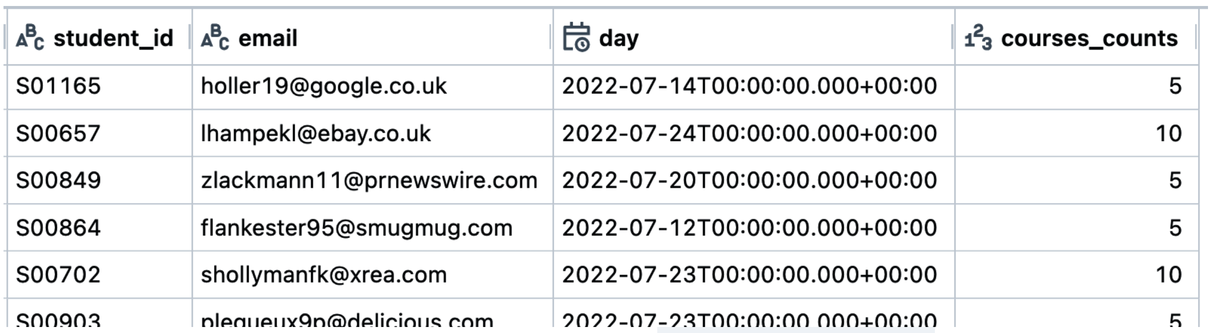
Structured Streaming assumes data is being appended only in the upstream tables. Once a table is updated or overwritten, it becomes invalid for streaming reads. Therefore, reading a stream from such a gold table is not supported. To alter this behavior, options like `skipChangeCommits` can be utilized, although they may come with other implications that need to be considered. See the Databricks documentation for more information.

By running this streaming query, our stream will process all available data in micro-batches and then stop automatically, thanks to the `availableNow` trigger option. This approach allows us to seamlessly integrate streaming and batch workloads within the same pipeline.

Now, we can inspect the aggregated data written into the `daily_student_courses` table:

```
%sql
SELECT * FROM daily_student_courses
```

Figure 5-31 displays the contents of our gold table, showcasing the daily enrollment statistics. You can observe that the students currently have course counts ranging between 5 and 10, reflecting the cumulative enrollment till now.



| student_id | email | day | courses_counts |
|---|---|---|---|
| S01165 | holler19@google.co.uk | 2022-07-14T00:00:00.000+00:00 | 5 |
| S00657 | lhampekl@ebay.co.uk | 2022-07-24T00:00:00.000+00:00 | 10 |
| S00849 | zlackmann11@prnewswire.com | 2022-07-20T00:00:00.000+00:00 | 5 |
| S00864 | flankester95@smugmug.com | 2022-07-12T00:00:00.000+00:00 | 5 |
| S00702 | shollymanfk@xrea.com | 2022-07-23T00:00:00.000+00:00 | 10 |
| S00903 | plequeux9p@delicious.com | 2022-07-23T00:00:00.000+00:00 | 5 |

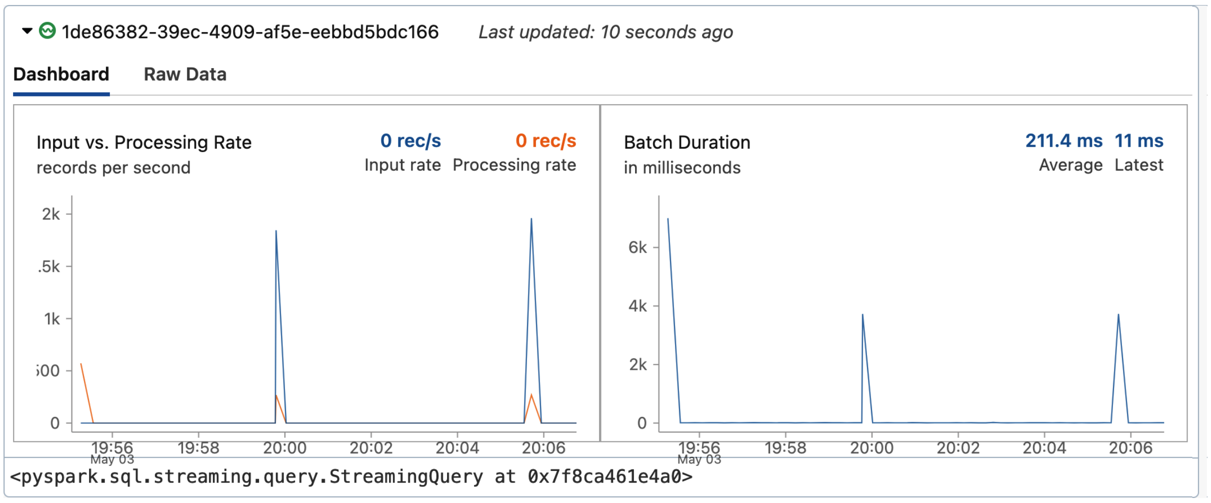
Figure 5-31. The result of querying the `daily_student_courses` table

Let's simulate the arrival of more new data by triggering the ingestion of another file into our source directory. This action initiates the propagation of data through our pipeline, from the bronze to the silver and gold layers:

```
load_new_data()

Output: Loading 06.json file to the school dataset
```

The new data will automatically propagate into both the bronze and silver layers as they maintain active continuous streams in place. Figure 5-32 illustrates the updated progress of the stream processing in the silver sayer after receiving the new data.

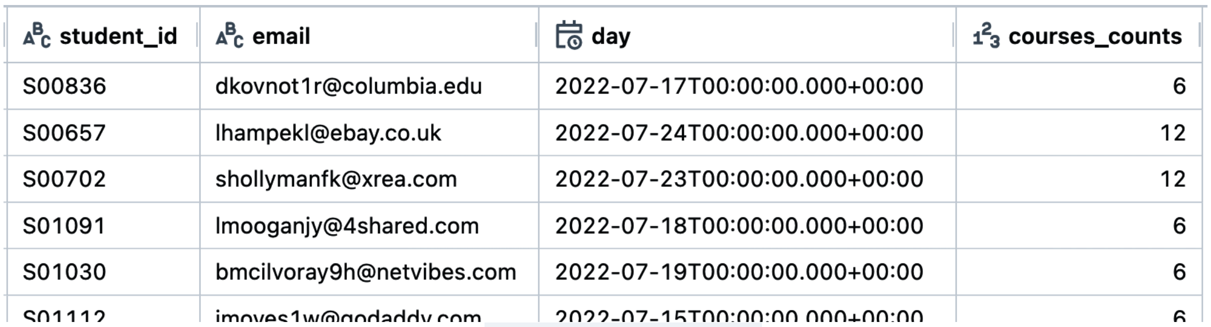


```
<pyspark.sql.streaming.query.StreamingQuery at 0x7f8ca461e4a0>
```

Figure 5-32. Stream processing in the silver layer after landing the new data file

However, for the gold layer, it’s necessary to explicitly rerun its streaming query to update the table. Remember, this query was configured as an incremental batch job using the `availableNow` trigger option.

Upon re-executing the streaming write query of our gold table, the newly ingested data is processed to reflect the latest changes. To confirm the successful update, we can again query the `daily_student_courses` table. Figure 5-33 illustrates the updated content of this gold table, showcasing students with an increased number of course enrollments.



| student_id | email | day | courses_counts |
|---|---|---|---|
| S00836 | dkovnot1r@columbia.edu | 2022-07-17T00:00:00.000+00:00 | 6 |
| S00657 | lhampekl@ebay.co.uk | 2022-07-24T00:00:00.000+00:00 | 12 |
| S00702 | shollymanfk@xrea.com | 2022-07-23T00:00:00.000+00:00 | 12 |
| S01091 | lmooganjy@4shared.com | 2022-07-18T00:00:00.000+00:00 | 6 |
| S01030 | bmcilvoray9h@netvibes.com | 2022-07-19T00:00:00.000+00:00 | 6 |
| S01112 | jmoves1w@godaddy.com | 2022-07-15T00:00:00.000+00:00 | 6 |

Figure 5-33. The result of querying the `daily_student_courses` table after processing the new data

## Stopping active streams

Finally, at the end of this demonstration, it’s important to ensure that all active streams in our notebook are properly terminated. This can be easily achieved by executing a loop that iterates through each active stream in the current Spark session and stops them:

```
for s in spark.streams.active:
   print("Stopping stream: " + s.id)
   s.stop()
   s.awaitTermination()

Stopping stream: 1de86382-39ec-4909-af5e-eebbd5bdc166
Stopping stream: f9086ea8-1091-4ba5-8682-476260b449ca
```

# Conclusion

In conclusion, the medallion architecture provides a structured and incremental approach to data processing, which is highly beneficial for modern data engineering tasks. By organizing data into distinct layers based on its level of refinement, this architecture enables you to efficiently process and analyze data, while ensuring data quality and accuracy. This makes it the ideal choice for building data pipelines in the lakehouse that can support a wide range of data-driven applications and analytics.

## Sample Exam Questions

### Conceptual Question

1. A data engineering team is working on a large-scale data pipeline for a global e-commerce platform. The platform collects vast amounts of customer transaction data, which is continuously landed into a cloud storage system in file format. The team needs to process this incoming data in near real-time, ensuring that all new files are ingested efficiently, without missing any records. The team decides to use Auto Loader for this task.

Based on this scenario, which of the following statements best describes how Auto Loader can help the data engineering team in this situation?

1. Auto Loader requires no computing resources, allowing users to process unlimited amounts of data without affecting performance.
2. Auto Loader automatically detects and processes new data files as they arrive in cloud storage, without reprocessing previously processed files.
3. Auto Loader reprocesses the entire set of files in cloud storage each time new data is added, which ensures that no file is missed.
4. Auto Loader is based on the `COPY INTO` command to ensure new files are detected and processed in real-time.
5. Auto Loader supports only batch processing, making it unsuitable for streaming or continuously updating data pipelines.

### Code-Based Question

2. A data engineer uses the following Structured Streaming query to process incoming orders and compute the total cost of each order, including tax. The processed data is then written to a table named `new_orders`:

```
( spark.table("orders")
        .withColumn("total_after_tax", col("total")+col("tax"))
```

```
            .writeStream
            .option("checkpointLocation", checkpointPath)
            .outputMode("append")
            .__________
            .table("new_orders"))
```

The engineer needs the query to execute multiple micro-batches to process all available data, and then stop automatically when there is no more data left to process.

Which of the following lines of code fills in the blank to achieve the desired outcome?

1. `trigger("micro-batches")`
2. `trigger(once=True)`
3. `trigger(processingTime="0 seconds")`
4. `trigger(micro-batches=True)`
5. `trigger(availableNow=True)`

The correct answers to these questions are listed in Appendix C.