**Trainee Q&A: How Auto Loader Handles Files Across Batches**

* **Mentor (IT Architect)**
* **Trainee (Junior Data Engineer)**

**Auto Loader** is a high-performance, scalable file ingestion tool provided by Databricks for ingesting new data files from cloud storage (like AWS S3, Azure Data Lake, or Google Cloud Storage) **automatically and incrementally** using **Databricks Structured Streaming**.

**Purpose:**

* Watches a directory for **new files only** — no need to reprocess old data.
* Supports **schema inference** and **evolution**.
* Handles **millions of files** efficiently using file notification services.
* Works with formats like **CSV, JSON, Parquet, Avro**, and more.
* Ideal for implementing **Bronze layer ingestion** in Delta Lake.

|  |  |
| --- | --- |
| Trainee | Mentor |
| Hey, I’ve been exploring Auto Loader in Databricks, and I’ve got a question. Let’s say we have 5 files to ingest — how exactly are they represented in the checkpoint directory?  Specifically in commits/ and sources/ — does sources track all 5 files or only the ones from the current micro-batch? | That’s a great question! Auto Loader organizes metadata in a very structured way — and understanding it will help you debug and optimize streaming jobs efficiently.  Let me break it down for you. There are **two possible ways** those 5 files could be processed, depending on when they arrive and your stream's trigger setting. |

|  |
| --- |
| Mentor  **Option A: All 5 files arrive together (one micro-batch)**  If all 5 files are discovered **before the next trigger fires**, Auto Loader groups them into a **single batch**.  You’ll see:   * commits/0 → indicating **batch 0 was committed** * sources/0 → listing all files seen in this batch:   json  CopyEdit  "seenFiles": [  "people\_1.csv",  "people\_2.csv",  "people\_3.csv",  "people\_4.csv",  "people\_5.csv"  ]  So in this case, sources/0 alone shows all 5 files — and they were all committed in batch 0. |

|  |
| --- |
| **Trainee:** Got it — so if all files arrive close together, one batch can handle all of them. What if the files arrive at different times? |

|  |
| --- |
| **Mentor**  **Option B: Files arrive at different times (multiple batches)**  Let’s say:   * 3 files arrive first, * and 2 more come in **10 seconds later**.   With a typical trigger interval (like every 10 seconds), Spark will process them in **two separate batches**:   * commits/0 → batch 0   + sources/0 → "seenFiles": ["people\_1.csv", "people\_2.csv", "people\_3.csv"] * commits/1 → batch 1   + sources/1 → "seenFiles": ["people\_4.csv", "people\_5.csv"]   So the files are **split across multiple sources/N files**, but all 5 are still tracked. |

|  |
| --- |
| **Trainee:** Oh, I see. So even though they’re split across batches, the total seen files still add up to 5. Makes sense! |

**Final Takeaway**

|  |  |
| --- | --- |
| **Folder** | **What it Reflects** |
| commits/ | Number of **micro-batches** (e.g., 0, 1, 2...) |
| sources/ | Tracks the **files seen in each batch** |
| Total seenFiles | Equals all files that were **actually processed** |

So, even if you process 5 files over 1, 2, or 10 batches, the total number of seenFiles across sources/ will still be 5.

|  |
| --- |
| **Trainee:** That clears it up perfectly! So it’s the **batches** that control the file grouping, not the number of files per se. |

|  |
| --- |
| **Mentor:** Exactly! It's **time-based batching**, not file-count-based. And once you understand how commits/ and sources/ align, it’s much easier to track and debug your Auto Loader pipelines. |

Appendix:

## ****Sample Auto Loader Code****

python

CopyEdit

from pyspark.sql.functions import current\_timestamp, input\_file\_name, upper, col

# Read data using Auto Loader

df = (

spark.readStream

.format("cloudFiles")

.option("cloudFiles.format", "csv") # File format

.option("cloudFiles.inferColumnTypes", "true") # Infer schema

.option("cloudFiles.schemaLocation", "/mnt/data/autoloader/schema/employee/") # Where schema is stored

.load("/mnt/data/autoloader/incoming/") # Input folder

)

# Apply transformations

df\_transformed = (

df.withColumn("department", upper(col("department")))

.withColumn("ingestion\_timestamp", current\_timestamp())

.withColumn("source\_file", input\_file\_name())

)

# Write to Delta Lake with checkpointing

(

df\_transformed.writeStream

.format("delta")

.option("checkpointLocation", "/mnt/data/autoloader/checkpoints/employee/") # Required

.outputMode("append")

.start("/mnt/data/bronze/employee") # Bronze table path

)

AutoLoader

|  |  |
| --- | --- |
| Config : Without Trigger | Config : With Trigger |
| df.writeStream \  .format("delta") \  .option("checkpointLocation", "/mnt/checkpoints/people") \  .start("/mnt/data/delta/bronze\_people") | df.writeStream \  .format("delta") \  .option("checkpointLocation", "/mnt/checkpoints/people") \  .trigger(processingTime="5 seconds") \  .start("/mnt/data/delta/bronze\_people") |
| **Behavior:**   |  |  | | --- | --- | | **Time** | **Action** | | 00:00 | Spark sees people\_1.csv and processes it in **batch 0** immediately | | 00:03 | Spark detects people\_2.csv and runs **batch 1** | | 00:06 | Spark sees people\_3.csv and starts **batch 2** | | ... | Spark polls continuously with **no delay** | | Behavior:  |  |  | | --- | --- | | **Time** | **Action** | | 00:00 | Spark sees people\_1.csv and starts **batch 0** | | 00:05 | No new files → batch runs, nothing processed | | 00:10 | Sees people\_2.csv and people\_3.csv (if they arrived) → **batch 1** processes them together | |
| ➡**3 batches for 3 files** ➡Latency = **as fast as Spark can respond** | ➡**Only 2 batches for 3 files** ➡Latency = **bounded by 5-second interval** |
| Comparison Table  |  |  |  | | --- | --- | --- | | **Feature** | **Without Trigger** | **With .trigger(processingTime="5s")** | | Trigger | Default (as fast as possible) | Fixed 5-second interval | | Batch Count | 3 batches (1 per file) | 2 batches (grouped by time) | | Latency | Lower (real-time) | Medium (5s delay max) | | Resource Efficiency | High CPU usage per file | More efficient grouping | | Control over behavior | No | Yes | | Use in Production | Can be noisy / expensive | More predictable | | |
| Summary  * **Without .trigger()** = lower latency, high responsiveness, but may create too many small batches. * **With .trigger()** = more control, better performance, lower cost at scale. * Choose based on:   + **Latency sensitivity** (alerts? dashboards?)   + **Cost and throughput**   + **File arrival pattern** | |

Final Note:

**Auto Loader** = smart, incremental ingestion from cloud storage with schema management and fault tolerance built-in. It's the recommended way to build the **Bronze layer** in modern data lakehouses using Delta Lake.