What is Auto Loader?

Auto Loader incrementally and efficiently processes new data files as they arrive in cloud storage without any additional setup.

It simplifies the setup of data pipelines for extracting, transforming, and loading (ETL/ELT) data into the Databricks Lakehouse.

How it works?

Auto Loader incrementally and efficiently processes new data files as they arrive in cloud storage. It provides a Structured Streaming source called **cloudFiles**.

Given an input directory path on the cloud file storage, the **cloudFiles** source automatically processes new files as they arrive, with the option of also processing existing files in that directory.

Key Features and Functionality

Auto Loader provides a Structured Streaming source called **cloudFiles** that monitors a specified cloud storage directory and automatically processes new files.

- **Incremental and Efficient Processing:** It detects and processes only new files that land in the source directory, making ingestion highly efficient and scalable, even with billions of files.
- **Structured Streaming Integration:** Auto Loader is built on top of Apache Spark Structured Streaming and offers an improved, scalable alternative to the standard file source.
- **Supported Sources:** It can load data from major cloud storage providers:
    - Amazon S3 (s3://)
    - Azure Data Lake Storage (ADLS Gen2, abfss://)
    - Google Cloud Storage (GCS, gs://)

- **Supported Formats:** It can ingest various file formats, including JSON, CSV, Parquet, AVRO, ORC, TEXT, XML, and BINARYFILE.
- **Exactly Once Guarantee:** It uses a scalable key-value store (RocksDB) in the checkpoint location to persist file metadata. This ensures that files are processed exactly once, even during stream restarts or failures.
- **Automatic Schema Inference and Evolution:** Auto Loader can automatically infer the schema of the ingested data and handle **schema drift** by detecting changes, notifying you, and rescuing data that might otherwise be ignored or lost.

File Discovery Modes

Auto Loader supports two primary modes for detecting new files:

1. **Directory Listing Mode (Default):**
    - Identifies new files by **listing the input directory**.
    - Best for smaller directories or when you want to get started quickly without extra cloud permissions.
2. **File Notification Mode:**
    - **Automatically sets up cloud notification services** (like AWS SQS/SNS or Azure Event Grid/Queue Storage) to subscribe to file events.
    - **More performant and scalable** for high-volume or very large input directories.
    - Reduces costs associated with repeated directory listings, but requires additional cloud permissions for setup.

---

can use Auto Loader for full load

Yes, you can use **Auto Loader for a full load** (or initial load/backfill) in addition to its primary use for incremental streaming.

How Auto Loader Handles Full Loads

Auto Loader is built on Apache Spark Structured Streaming, which is designed for continuous, incremental processing. However, you can configure the stream to process all existing files in the source directory when the stream starts.

1. **Including Existing Files:** (By default), **when you first start an Auto Loader stream, it will process all existing files in the specified path**. You don't usually need a special option for this initial behaviour unless you explicitly want to skip existing files.
    - To explicitly ensure existing files are included on the first run, you can set the option:

      .option("cloudFiles.includeExistingFiles", "true")
2. **Using Trigger.AvailableNow (Incremental Batch):**
    - For a one-time full load, or a large backfill that you want to execute as an efficient batch job,
    - **Databricks recommends using the Trigger.AvailableNow setting in the writeStream.**

    # Example for a one-time full load/backfill

    ```
    (df.writeStream
      .option("checkpointLocation", "path/to/checkpoint")
      .trigger(availableNow=True) # Processes all available data and then stops
      .toTable("target_table"))
    ```

    This trigger instructs the stream to process all files that are currently available in the source directory (as discovered by Auto Loader) and then shut down. Subsequent runs will only process newly arrived files since the last successful completion.

In summary, Auto Loader's strength lies in its ability to start with an efficient full load and seamlessly transition to continuous, incremental processing without any change in your code.

---

1. The Role of AvailableNow (Incremental Batch)

The Trigger.AvailableNow setting is used to process a defined, finite chunk of data.

- First Run (Backfill/Full Load): The job starts, processes all data available in the source (e.g., all files in the S3 folder) up to that moment, writes the results, updates the checkpoint location to mark the last file processed, and then stops.
- Subsequent Runs (Incremental Batches): You re-run the exact same query. Because the job reuses the previous checkpoint location, it knows exactly where it left off. It then performs the following steps:
    1. It checks the source (the S3 folder) for all data that has arrived since the last checkpoint.
    2. It processes this *new incremental data* (the next batch).
    3. It updates the checkpoint with the new offset.
    4. It stops again.

This pattern is often used for scheduled jobs (e.g., an ETL pipeline scheduled to run every hour) where you want to efficiently pick up all new data and then shut down the compute resources immediately.

---

2. Alternatives for Continuous Streaming

If your requirement is for continuous, low-latency processing where you do *not* want the job to stop, you would use a different trigger:

Trigger Mode	Behavior for Incremental Data	Primary Use Case
Default Trigger	The stream never stops. It automatically and continuously checks for new data and processes it in tiny batches (micro-batches) as quickly as possible.	Real-time processing and ingestion pipelines.
Trigger.ProcessingTime('X time')	The stream never stops. It wakes up every 'X time' (e.g., every 5 minutes) to check for and process all available new data.	Scheduled, near real-time processing where a slight delay is acceptable.

Mode

You should choose the Auto Loader file detection mode based on the volume of files you expect, the latency requirements, and your cloud permission flexibility.

The two modes are:

1. **Directory Listing Mode (Default)**

2. **File Notification Mode**

Directory Listing Mode
This is the simpler, default mode and works by having Databricks run a command to list the files in the input directory.

When to Use	Why
Small-to-Medium File Volume	It's efficient for input directories with fewer files or when files don't arrive very frequently.
Simple Setup / Limited Permissions	It requires no extra cloud permissions or setup outside of Databricks (just read access to the storage). You can start quickly.
Highly-Partitioned Data	If your files are organized into a large number of date-time partitions (e.g., millions of directories), this mode can become slow because it has to list all of them to find the new data. *However*, Databricks has optimizations like Incremental Listing to try and mitigate this.
Low Latency Is Not Critical	The time it takes to scan the directory can introduce slight delays, making it less ideal for near-real-time needs.

File Notification Mode
This mode is event-driven. Databricks automatically sets up native cloud services (like AWS SQS/SNS or Azure Event Grid/Queue Storage) that listen for file-creation events from the storage container. The cloud service pushes a notification to a queue when a new file arrives, which the Auto Loader stream then reads.

When to Use	Why
High File Volume (Millions per hour)	It's highly scalable and performant because it doesn't need to scan the entire directory structure to find new files. It simply reads a short list of new file events.
Near-Real-Time (Low Latency)	File event notifications are near-instant, offering the lowest ingestion latency.
Cost-Efficiency	This mode can be more cost-efficient for large-scale ingestion compared to the cloud API charges for constantly listing directories.
Complex Setup / Ample Permissions	It requires additional cloud permissions for Databricks to automatically create and manage the necessary event and queue services in your cloud account.

Summary

Factor	Directory Listing Mode	File Notification Mode
Scalability	**Lower (Can be slow with billions of files/directories)**	**Higher (Scales to millions of files per hour)**
Latency	**Higher (Due to directory scanning time)**	**Lower (Near-instant notification)**
Setup Complexity	**Low (Simple configuration)**	**High (Requires additional cloud permissions)**
Cost	**Can be higher due to cloud API listing costs at scale**	**Generally lower for high-volume scenarios**
Recommendation	**Start with this mode unless you hit performance/scalability limits.**	**Use for production-grade, high-volume or low-latency pipelines.**

Auto Loader has a wide range of configuration options that control everything from performance to data quality.

The key options fall into three main categories:

1. **File Discovery and Backfill (Performance & Cost)**
2. **Schema Inference and Evolution (Data Robustness)**
3. **Rate Limiting and Processing (Control)**

---

1. File Discovery and Backfill Options

These options control how Auto Loader finds files and handles historical data.

Option	Description	When to Use
cloudFiles.**format** **(Required)**	Specifies the file format of the source data.	Must be set for formats like json, csv, parquet, avro, orc, text, or binaryFile.

Option	Description	When to Use
cloudFiles.**useNotifications**	File Detection Mode Selector (Set to true or false).	Set to true to enable the File Notification Mode for high volume/low latency. (Requires cloud permissions).
cloudFiles.**backfillInterval**	Triggers an asynchronous backfill scan at a given interval (e.g., '1 day').	Use on long-running streams to periodically check for files that might have been missed by event notifications or to quickly process files uploaded during a stream downtime.
cloudFiles.includeExistingFiles	Whether to process files that already exist in the directory when the stream starts for the first time.	Set to 'true' to perform a full initial load (the default behavior). Set to 'false' to only process new files.
cloudFiles.**maxFileAge**	Ignores files older than the specified time (e.g., '7 days', '1 month').	Use to limit ingestion to recent data, saving on discovery time and costs. Files older than this are ignored, even on a backfill.
pathGlobFilter	A glob pattern (wildcard) to filter file paths.	Use to include or exclude specific files based on their name or path (e.g., *.csv or 2024/*/*/*.json).

---

2. Schema Inference and Evolution Options
These options manage how Auto Loader handles the structure of your data.

Option	Description	When to Use
cloudFiles.**schemaLocation** **(Required for inference)**	The checkpoint location where the inferred	Mandatory for automatic schema

Option	Description	When to Use
	schema is stored and tracked for changes.	inference and evolution.
cloudFiles.**inferColumnTypes**	When inferring the schema, attempts to determine specific column data types (e.g., integer, timestamp) instead of defaulting to string.	Use to get a more precise initial schema. (Default: false for schemaless formats like JSON/CSV).
cloudFiles.**schemaEvolutionMode**	Defines how Auto Loader reacts to schema changes (new columns, type changes).	Crucial setting. The common modes are:
	-addNewColumns (Recommended)	Automatically adds new columns to the end of the DataFrame and continues processing.
	- failOnNewColumns	Stops the stream if a new column is detected, requiring a manual schema update.
	- rescue	Does not evolve the schema, but places all new or unexpected fields into a special column named _rescued_data to prevent data loss.
cloudFiles.rescueDataColumn	The name of the column where malformed or stray data is placed. (Default: _rescued_data).	Use to rename the default "bad record" column.

3. Rate Limiting and Processing Options

These options control the batch size for each micro-batch. They are primarily used when running the stream in an Incremental Batch (Trigger.AvailableNow) or a continuous processing mode.

Option	Description	When to Use
cloudFiles.**maxFilesPerTrigger**	The hard maximum number of files to process in a single micro-batch.	Use to control the batch size, making micro-batches smaller and faster, which can improve throughput and job stability.
cloudFiles.**maxBytesPerTrigger**	The soft maximum number of bytes to process in a single micro-batch (e.g., '10g').	Use to limit the volume of data processed, especially when files are large or have highly variable sizes.
cloudFiles.**cleanSource**	Whether to delete or move source files after they have been processed and written to the target (Delta Lake).	Set to 'DELETE' or 'MOVE' ('OFF' is the default) only when you are certain the source files are no longer needed. Requires specific cloud permissions.

```
df = spark.readStream.format("cloudFiles") \
  .option("cloudFiles.format", "json") \
  .option("cloudFiles.schemaLocation", "/mnt/datalake/autoloader/schema") \
  .option("cloudFiles.schemaEvolutionMode", "addNewColumns") \
  .option("cloudFiles.includeExistingFiles", "true") \
  .option("cloudFiles.maxFilesPerTrigger", 5000) \
  .load("/mnt/datalake/raw/events")

(df.writeStream
  .option("checkpointLocation", "/mnt/datalake/autoloader/checkpoint")
  .trigger(availableNow=True) # or processingTime='1 minute'
  .toTable("bronze_events_table"))
```

Schema Evolution and Handling Data Quality, Production Optimization and Monitoring, and integrating it with Delta Live Tables (DLT) for end-to-end pipelines.

Here are the key areas you should learn next:

1. Schema Evolution and Data Quality
Auto Loader has powerful features for handling changes in your incoming data, which is crucial in real-world scenarios.

- **Schema Evolution Modes: Understand the different modes and when to use them:**
  - **addNewColumns (Default):** The stream fails, then the schema evolves to include new columns, and the stream is typically restarted. This is generally used when you want a controlled failure to acknowledge the change.
  - **rescue:** New columns and data type mismatches are collected into a special column (default is _rescued_data), and the stream *doesn't* fail. This is great for fault-tolerance in the "Bronze" layer of a Lakehouse architecture.
  - **failOnNewColumns:** The stream fails permanently if a new column is detected, requiring a manual fix.
- **Schema Hints:** Learn how to use schemaHints to override the inferred schema for specific columns, especially for complex or nested types that might be misidentified (like forcing a string to a specific numeric type).
- **Handling Nested Data:** For formats like JSON or Avro, understand how to work with nested objects, arrays, and inferring their schemas, as well as how they relate to schema evolution.
- **Data Quality with Delta Live Tables (DLT) Expectations:** When combining Auto Loader with DLT, you can define Expectations to enforce data quality (e.g., NOT NULL, value range checks). Learn how to configure DLT to *drop*, *quarantine*, or *fail* records/batches that violate these rules.

---

2. Production Optimization and Monitoring
To use Auto Loader in a reliable and cost-efficient production environment, tuning and observability are essential.

- **File Detection Modes: Auto Loader supports two primary modes for finding new files:**
  - **Directory Listing (Default): Periodically lists the input directory. Best for a smaller number of files or when file-level event notifications aren't supported.**
  - **File Notification: Uses native cloud services (like AWS S3 Event Notifications or Azure Event Grid/Notification Hub) to get notified when a new file lands. This is more scalable and cost-effective for high-volume, high-velocity data.**
- **Tuning and Rate Limiting:**
  - **cloudFiles.maxFilesPerTrigger / cloudFiles.maxBytesPerTrigger: Control the maximum number of files or data size processed in each micro-batch to manage load and keep latency predictable.**

    - **cloudFiles.maxFileAge: A cost control mechanism to ignore file events older than a specified time, preventing the state store from growing indefinitely. Use with caution to avoid missing data.**
- **Exactly-Once Guarantees and Checkpointing: Understand that Auto Loader uses a RocksDB state store in the checkpointLocation to track which files have been processed, ensuring each file is processed exactly once, even during failures and restarts. The checkpoint location is critical and must be managed carefully.**
- **Monitoring and Metrics: Learn how to access and interpret the stream metrics reported to the Spark Streaming Query Listener, specifically:**
    - **numFilesOutstanding**
    - **numBytesOutstanding**
    - **These metrics help you gauge the backlog size of your incoming data stream.**

---

3. Advanced Integration Patterns 🖇️

Auto Loader is often the first step in a Lakehouse pattern, typically ingesting data into the Bronze layer. You should learn how it fits into the broader data pipeline:

- **Delta Live Tables (DLT): This is the highly recommended, declarative framework on Databricks for building data pipelines, and it integrates natively with Auto Loader (using the STREAM read_files(...) SQL function or dlt.read() in Python). DLT simplifies schema evolution, error handling, and deployment.**
- **Upserts (Merge Operations): Auto Loader only handles incremental ingestion (adding new files). To handle updates or deletes to your target Delta table, you need to use the foreachBatch function with a DeltaTable.merge() operation *after* Auto Loader reads the micro-batch. This is a common pattern for the Silver layer.**
- **Handling Late-Arriving Data: Understand how Auto Loader's file tracking handles files that arrive out of order or much later than expected, and how configuring cloudFiles.backfillInterval can be used as a fallback for strict data completeness requirements.**