Last updated on ***Nov 5, 2025***

Common data loading patterns

Auto Loader simplifies a number of common data ingestion tasks. This quick reference provides examples for several popular patterns.

Ingest data from cloud object storage as variant

Auto Loader can load all data from the supported file sources as a single `VARIANT` column in a target table. Because `VARIANT` is flexible to schema and type changes and maintains case sensitivity and `NULL` values present in the data source, this pattern is robust to most ingestion scenarios. For details, see Ingest data from cloud object storage as variant.

Filtering directories or files using glob patterns

Glob patterns can be used for filtering directories and files when provided in the path.

Pattern	Description
`?`	Matches any single character
`*`	Matches zero or more characters
`[abc]`	Matches a single character from character set {a,b,c}.
`[a-z]`	Matches a single character from the character range {a...z}.

Pattern	Description
`[^a]`	Matches a single character that is not from character set or range {a}. Note that the `^` character must occur immediately to the right of the opening bracket.
`{ab,cd}`	Matches a string from the string set {ab, cd}.
`{ab,c{de, fh}}`	Matches a string from the string set {ab, cde, cfh}.

Use the `path` for providing prefix patterns, for example:

Python Scala

Python

```
df = spark.readStream.format("cloudFiles") \
  .option("cloudFiles.format", <format>) \
  .schema(schema) \
  .load("<base-path>/*/files")
```

IMPORTANT

You need to use the option `pathGlobFilter` for explicitly providing suffix patterns. The `path` only provides a prefix filter.

For example, if you would like to parse only `png` files in a directory that contains files with different suffixes, you can do:

Python Scala

Python

```
df = spark.readStream.format("cloudFiles") \
  .option("cloudFiles.format", "binaryFile") \
  .option("pathGlobfilter", "*.png") \
  .load(<base-path>)
```

NOTE

The default globbing behavior of Auto Loader is different than the default behavior of other Spark file sources. Add `.option("cloudFiles.useStrictGlobber", "true")` to your read to use globbing that matches default Spark behavior against file sources. See the following table for more on globbing:

Pattern	File path	Default globber	Strict globber
/a/b	/a/b/c/file.txt	*Yes*	*Yes*
/a/b	/a/b_dir/c/file.txt	*No*	*No*
/a/b	/a/b.txt	*No*	*No*
/a/b/	/a/b.txt	*No*	*No*
/a/*/c/	/a/b/c/file.txt	*Yes*	*Yes*
/a/*/c/	/a/b/c/d/file.txt	*Yes*	*Yes*
/a/*/c/	/a/b/x/y/c/file.txt	*Yes*	*No*
/a/*/c	/a/b/c_file.txt	*Yes*	*No*
/a/*/c/	/a/b/c_file.txt	*Yes*	*No*
/a/*/c/	/a/*/cookie/file.txt	*Yes*	*No*
/a/b*	/a/b.txt	*Yes*	*Yes*
/a/b*	/a/b/file.txt	*Yes*	*Yes*

Pattern	File path	Default globber	Strict globber
/a/{0.txt,1.txt}	/a/0.txt	*Yes*	*Yes*
/a/*/{0.txt,1.txt}	/a/0.txt	*No*	*No*
/a/b/[cde-h]/i/	/a/b/c/i/file.txt	*Yes*	*Yes*

Enable easy ETL

An easy way to get your data into Delta Lake without losing any data is to use the following pattern and enabling schema inference with Auto Loader. Databricks recommends running the following code in a Databricks job for it to automatically restart your stream when the schema of your source data changes. By default, the schema is inferred as string types, any parsing errors (there should be none if everything remains as a string) will go to `_rescued_data`, and any new columns will fail the stream and evolve the schema.

Python **Scala**

Python

```
spark.readStream.format("cloudFiles") \
  .option("cloudFiles.format", "json") \
  .option("cloudFiles.schemaLocation", "<path-to-schema-location>") \
  .load("<path-to-source-data>") \
  .writeStream \
  .option("mergeSchema", "true") \
  .option("checkpointLocation", "<path-to-checkpoint>") \
  .start("<path_to_target")
```

Prevent data loss in well-structured data

When you know your schema, but want to know whenever you receive unexpected data, Databricks recommends using the `rescuedDataColumn`.

Python Scala

Python

```
spark.readStream.format("cloudFiles") \
  .schema(expected_schema) \
  .option("cloudFiles.format", "json") \
  # will collect all new fields as well as data type mismatches in _rescued_data
  .option("cloudFiles.schemaEvolutionMode", "rescue") \
  .load("<path-to-source-data>") \
  .writeStream \
  .option("checkpointLocation", "<path-to-checkpoint>") \
  .start("<path_to_target")
```

If you want your stream to stop processing if a new field is introduced that doesn't match your schema, you can add:

Python

```
.option("cloudFiles.schemaEvolutionMode", "failOnNewColumns")
```

Enable flexible semi-structured data pipelines

When you're receiving data from a vendor that introduces new columns to the information they provide, you may not be aware of exactly when they do it, or you may not have the bandwidth to update your data pipeline. You can now leverage schema evolution to restart the stream and let Auto Loader update the inferred schema automatically. You can also leverage `schemaHints` for some of the “schemaless” fields that the vendor may be providing.

Python Scala

Python

Ask Assistant

```
spark.readStream.format("cloudFiles") \
  .option("cloudFiles.format", "json") \
  # will ensure that the headers column gets processed as a map
  .option("cloudFiles.schemaHints",
          "headers map<string,string>, statusCode SHORT") \
  .load("/api/requests") \
  .writeStream \
  .option("mergeSchema", "true") \
  .option("checkpointLocation", "<path-to-checkpoint>") \
  .start("<path_to_target")
```

Transform nested JSON data

Because Auto Loader infers the top level JSON columns as strings, you can be left with nested JSON objects that require further transformations. You can use the semi-structured data access APIs to further transform complex JSON content.

Python Scala

Python

```
spark.readStream.format("cloudFiles") \
  .option("cloudFiles.format", "json") \
  # The schema location directory keeps track of your data schema over time
  .option("cloudFiles.schemaLocation", "<path-to-checkpoint>") \
  .load("<source-data-with-nested-json>") \
  .selectExpr(
    "*",
    "tags:page.name",    # extracts {"tags":{"page":{"name":...}}}
    "tags:page.id::int", # extracts {"tags":{"page":{"id":...}}} and casts to int
    "tags:eventType"     # extracts {"tags":{"eventType":...}}
  )
```

Infer nested JSON data

When you have nested data, you can use the `cloudFiles.inferColumnTypes` option to infer the nested structure of your data and other column types.

Python Scala

Python

```
spark.readStream.format("cloudFiles") \
  .option("cloudFiles.format", "json") \
  # The schema location directory keeps track of your data schema over time
  .option("cloudFiles.schemaLocation", "<path-to-checkpoint>") \
  .option("cloudFiles.inferColumnTypes", "true") \
  .load("<source-data-with-nested-json>")
```

Load CSV files without headers

Python Scala

Python

```
df = spark.readStream.format("cloudFiles") \
  .option("cloudFiles.format", "csv") \
  .option("rescuedDataColumn", "_rescued_data") \ # makes sure that you don't lose data
  .schema(<schema>) \ # provide a schema here for the files
  .load(<path>)
```

Enforce a schema on CSV files with headers

Python Scala

Python

```
df = spark.readStream.format("cloudFiles") \
  .option("cloudFiles.format", "csv") \
```

```
  .option("header", "true") \
  .option("rescuedDataColumn", "_rescued_data") \ # makes sure that you don't lose data
  .schema(<schema>) \ # provide a schema here for the files
  .load(<path>)
```

Ingest image or binary data to Delta Lake for ML

Once the data is stored in Delta Lake, you can run distributed inference on the data. See Perform distributed inference using pandas UDF.

Python Scala

Python

```
spark.readStream.format("cloudFiles") \
  .option("cloudFiles.format", "binaryFile") \
  .load("<path-to-source-data>") \
  .writeStream \
  .option("checkpointLocation", "<path-to-checkpoint>") \
  .start("<path_to_target")
```

Auto Loader syntax for Lakeflow Spark Declarative Pipelines

Lakeflow Spark Declarative Pipelines provides slightly modified Python syntax for Auto Loader and adds SQL support for Auto Loader.

The following examples use Auto Loader to create datasets from CSV and JSON files:

Python **SQL**

SQL

```
CREATE OR REFRESH STREAMING TABLE customers
AS SELECT * FROM STREAM read_files(
  "/databricks-datasets/retail-org/customers/",
  format => "csv"
)

CREATE OR REFRESH STREAMING TABLE sales_orders_raw
AS SELECT * FROM STREAM read_files(
  "/databricks-datasets/retail-org/sales_orders/",
  format => "json")
```

You can use supported format options for Auto Loader. Options for `read_files` are key-value pairs. For details on supported formats and options, see Options.

For example:

SQL

```
CREATE OR REFRESH STREAMING TABLE my_table
AS SELECT *
  FROM STREAM read_files(
    "/Volumes/my_volume/path/to/files/*",
    option-key => option-value,
    ...
  )
```

The following example reads data from tab-delimited CSV files with a header:

SQL

```
CREATE OR REFRESH STREAMING TABLE customers
AS SELECT * FROM STREAM read_files(
  "/databricks-datasets/retail-org/customers/",
  format => "csv",
  delimiter => "\t",
  header => "true"
)
```

You can use the `schema` to specify the format manually; you must specify the `schema` for formats that do not support schema inference:

Python SQL

SQL

```
CREATE OR REFRESH STREAMING TABLE wiki_raw
AS SELECT *
FROM STREAM read_files(
  "/databricks-datasets/wikipedia-datasets/data-001/en_wikipedia/articles-only-parquet",
  format => "parquet",
  schema => "title STRING, id INT, revisionId INT, revisionTimestamp TIMESTAMP,
revisionUsername STRING, revisionUsernameId INT, text STRING"
)
```

NOTE

Lakeflow Spark Declarative Pipelines automatically configures and manages the schema and checkpoint directories when using Auto Loader to read files. However, if you manually configure either of these directories, performing a full refresh does not affect the contents of the configured directories. Databricks recommends using the automatically configured directories to avoid unexpected side effects during processing.