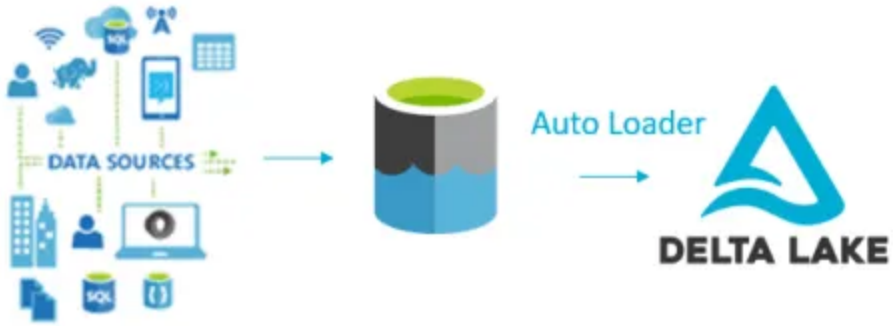
Databricks Autoloader Cookbook

Rahul Singha Follow 10 min read · Mar 16, 2023



In this article, we are going to discuss the following topics:

1. How Autoloader handles empty files and file names starting with an underscore
2. When to use the compression codec in Autoloader and what are the best practices for compressed files and various file formats
3. modifiedAfter and modifiedBefore in Autoloader

4. partitionColumns

5. overWrites

6. ignoreMissingFiles

7. pathGlobFilter

8. Moving Autoloader Job from one place to another Workspace to Another

In Databricks, when data is streamed using an autoloader, it should be made sure that the file names must not begin with an underscore '_', Otherwise, files will be ignored by the autoloader.

This can be explained with an example. Initially, three CSV files are kept in a directory in Azure Data Lake Storage, a non-empty CSV (sample1.csv), an empty CSV (empty_file.csv) and a non-empty CSV file whose name starts with an underscore '_' (_sample2.csv). This can be verified using the dbutils.fs.ls.

```
1 display(dbutils.fs.ls("abfss://<container>@<storage_acc>.dfs.core.windows.net/directory"))
```

▸ (2) Spark Jobs

Table +

	path	name	size	modificationTime
1	abfss://<container>@<storage_account>.dfs.core.windows.net/directory/_sample2.csv	_sample2.csv	3240604	1674124682000
2	abfss://<container>@<storage_account>.dfs.core.windows.net/directory/empty_file.csv	empty_file.csv	0	1674124680000
3	abfss://<container>@<storage_account>.dfs.core.windows.net/directory/sample1.csv	sample1.csv	3240604	1674124972000

Showing all 3 rows. | 1.10 seconds runtime

Autoloader is then used to load the files from the directory.

```
spark.readStream
      .format("cloudFiles")\
      .option("cloudFiles.format", "csv")\
      .option("cloudFiles.schemaLocation", "<schema_location>")\
      .option("cloudFiles.useIncrementalListing", "auto")\
      .load("abfss://<container>@<storage_account>.dfs.core.windows.net/directory
      .writeStream
      .option("checkpointLocation", "<path_to_checkpoint_location>")\
      .trigger(availableNow=True)\
      .table("table_name")
```

The cloud_files_state function of Databricks, which keeps track of the file-level state of an autoloader cloud-file source, confirmed that the autoloader processed only two files, non-empty CSV (sample1.csv) and empty CSV (empty_file.csv). The autoloader ignored the file ‘ _sample2.csv ’, whose name starts with an ‘_’.

```
1  %sql
2  select * from cloud_files_state("dbfs:/user/<path>/<to>/<checkpoint>");
```

▸ (1) Spark Jobs

▸ _sqldf: pyspark.sql.dataframe.DataFrame = [path: string, size: long ... 4 more fields]

Table +

	path	size	create_time	discovery_time	commit_time	archive_time
1	<container>@<storage_account>/directory/empty_file.csv	0	2023-01-19T10:38:00.000+0000	null	null	null
2	<container>@<storage_account>/directory/sample1.csv	3240604	2023-01-19T10:42:52.000+0000	null	null	null

2. When to use the compression codec in Autoloader and what are the best practices for compressed files and various file formats

This article provides a detailed comparison of the processing time of Autoloader for a variety of file types in Databricks.

The codec used is " .option('io.compression.codecs', 'nl.basjes.hadoop.io.compress.SplittableGzipCodec') ".

In order to use the codec in the Databricks notebook, the appropriate library must be installed in the cluster. Go to cluster details > Libraries tab > Install new > Maven > Search packages > Enter the name of the package that is "splittablegzip" and install.

The cluster configuration used for this comparison is given below.

DBR version 11.3 LTS

Driver and Worker Type: i3.2xlarge 61GB Memory, 8 cores

Number of workers 8

Autoscaling disabled

Used photon acceleration

For convention, the same data is converted to various file formats and compression and is stored in 2 files only.

The size of the file varies significantly for different file formats when the same data is stored.

Case 1: CSV and CSV GZIP

CSV: 15.1 GB x 2

CSV Gzip: 6.86 GB x 2

When processed, the following observations are noted.

File Type	Using Schema	Using Codec	Processing time*
CSV	✘	NA	18.96 mins
CSV	✔	NA	1.35 mins
CSV Gzip file	✘	✘	21.26 mins
CSV Gzip file	✔	✘	9.88 mins
CSV Gzip file	✘	✔	21.87 mins
CSV Gzip file	✔	✔	8.68 mins

Case 2: JSON and JSON GZIP

JSON: 20.88 GB x 2

JSON Gzip: 8.25 GB x 2

When processed, the following observations are note

File Type	Using Schema	Using Codec	Processing time
JSON file	✘	NA	3.79 mins
JSON file	✔	NA	1.85 mins
JSON Gzip file	✘	✘	41.47 mins
JSON Gzip file	✔	✘	17.96 mins
JSON Gzip file	✘	✔	25.91 mins
JSON Gzip file	✔	✔	4.46 mins

Case 3: TSV and TSV Gzip

TSV: 2.74 GB x 12 files

TSV Gzip: 1.21 GB x 12 files

When processed, the following observations are noted.

File Type	Using Schema	Using Codec	Processing time
TSV file	✘	NA	4.53 mins
TSV file	✔	NA	2.31 mins
TSV Gzip file	✘	✘	4.47 mins
TSV Gzip file	✔	✘	2.47 mins
TSV Gzip file	✘	✔	4.39 mins
TSV Gzip file	✔	✔	3.85 mins

Case 4: Parquet and Snappy

Parquet: 6.22 GB x 2 files

Snappy: 6.16 GB x 2 files

When processed, the following observations are noted.

File Type	Using Schema	Processing time
Parquet file	✘	4.38 mins
Parquet file	✔	3.87 mins
Parquet snappy file	✘	53 secs
Parquet snappy file	✔	27 secs

Conclusion:

1. Parquet is faster and the size of Parquet compression is less as compared to other file formats when the same data is stored.
2. Assigning schema manually also will improve the performance because it does not do the schema inference with the huge set of files. Thus, performance will be enhanced. But at the same time, the schema evolution, which is an advantage of Autoloader, will be lost.
3. The codec works with JSON Gzip compression and helps to process the data faster in Databricks.
4. The codec does not work with CSV and TSV Gzip compression.
5. It is recommended to use binary format like parquet for faster processing.

3. modifiedAfter and modifiedBefore in Autoloader

modifiedBefore and modifiedAfter are options that can be applied together or separately in order to achieve greater granularity over which files may load during a Spark batch query.

- modifiedBefore: an optional timestamp to only include files with modification times occurring before the specified time.
- modifiedAfter: an optional timestamp to only include files with modification times occurring after the specified time.

The provided timestamp must be in the following format: YYYY-MM-DDTHH:mm:ss (e.g. 2020–06–01T13:00:00). When a timezone option is not provided, the timestamps will be interpreted according to the Spark session timezone (spark.sql.session.timeZone). The default value is None.

This modification time can be converted from unix to utc using the following command which is the required timestamp format of modifiedAfter and modefiedBefore.

```
import datetime
def unix_to_utc(unix_timestamp):
    return datetime.datetime.utcfromtimestamp(unix_timestamp).strftime('%Y-%m-%d
```

The following line of code can be used to load files using autoloader with modifiedAfter option and write the data to a new delta table

```
spark.readStream
     .format("cloudFiles")
```

```
        .option("cloudFiles.format", "csv")
        .option("modifiedAfter",str(unix_to_utc((<unix_time>/1000)))
        .option("cloudFiles.schemaLocation", "<schema_location>")
        .option("cloudFiles.useIncrementalListing", "auto")
        .load("<location_of_files>")
        .writeStream
        .option("checkpointLocation", "<checkpoint_location>")
        .trigger(availableNow=True)
        .table("table_name")
```

This can be easily understood with the help of an example.

A directory contains 4 files with different modification time stamps which can be seen using **dbutils.fs.ls** command.

```
display(dbutils.fs.ls("directory_path"))
```

▸ (4) Spark Jobs

Table +

	path	name	size	modificationTime
1	file1_path/file1.csv	file1.csv	23	1675069906000
2	file2_path/file2.csv	file2.csv	23	1675069985000
3	file3_path/file3.csv	file3.csv	23	1675070015000
4	file4_path/file4.csv	file4.csv	23	1675070033000

Showing all 4 rows. | 1.08 seconds runtime

When modifiedAfter is used with a timestamp equal to the time stamp of file2, data of only those files where the modification time is strictly after the modifiedAfter timestamps are read. This can be observed by displaying the metadata column.

```
df_mA = spark.readStream\
          .format("cloudFiles")\
          .option("cloudFiles.format", "csv")\
          .option("modifiedAfter",str(unix_to_utc(1675069985000/1000)))\
          .option("cloudFiles.schemaLocation", "<schema_location>")\
          .option("cloudFiles.useIncrementalListing", "auto")\
          .load("<files_location>")\
          .select("*", "_metadata").display()
```

▸ (2) Spark Jobs

▸ display_query_5 (id: 3e6d4a10-5633-4aea-926a-73d57cccd86e) *Last updated: 19 minutes ago*

Table +

	Name	Data	_rescued_data	_metadata
1	person_4	264	null	▸ {"file_path": '<file_path of file_ 4>/file4.csv", "file_name": "file4.csv", "file_size": 23, "file_modification_time": "2023-01-30T09:13:53.000+0000"}
2	person_3	636	null	▸ {"file_path": '<file_path of file_3>/file3.csv", "file_name": "file3.csv", "file_size": 23, "file_modification_time": "2023-01-30T09:13:35.000+0000"}

Showing all 2 rows.

These options can be used together to get data of files with modification time within a certain interval of time bounded by the modifiedAfter and modifiedBefore. Here, modifiedAfter is used with a timestamp equal to the timestamp of file1 and modifiedBefore is used with a timestamp equal to the timestamp of file4. Thus, only file2 and file3 are loaded.

```
df = spark.readStream\
          .format("cloudFiles")\
          .option("cloudFiles.format", "csv")\
          .option("modifiedAfter",str(unix_to_utc(1675069906000/1000)))\
          .option("modifiedBefore",str(unix_to_utc(1675070033000/1000)))\
          .option("cloudFiles.schemaLocation", "<schema_location>")\
          .option("cloudFiles.useIncrementalListing", "auto")\
          .load("<location_of_files>")\
          .select("*", "_metadata").display()
```

▸ (1) Spark Jobs

▸ display_query_2 (id: 72aeba36-835e-4068-a68f-dd32e8ac63ea) *Last updated: 3 minutes ago*

Table +

	Name	Data	_rescued_data	_metadata
1	person_2	720	null	▸ {"<file_path_of_file2>/file2.csv", "file_name": "file2.csv", "file_size": 23, "file_modification_time": "2023-01-30T09:13:05.000+0000"}
2	person_3	636	null	▸ {"<file_path_of_file3>/file3.csv", "file_name": "file3.csv", "file_size": 23, "file_modification_time": "2023-01-30T09:13:35.000+0000"}

Showing all 2 rows.

4. partitionColumns inference

Partition columns are columns that are inferred from the directory structure of the files.

They are key-value pairs combined by an equality sign such as <base_path>/a=x/b=y/c=z/file.format.

For example, if directories are created in this order:

root_path>year=2022>month=january>day=21>file1.csv

root_path>year=2023>month=july>day=20>file2.csv

root_path>year=2023>month=october>day=12>file6.csv

The partition columns in the above example are year, month, and date. By default, these columns will be automatically added to the schema if schema inference is used and the <root_path> is provided to load data from. In case the partition column does not appear, the following code can be used to specify the partition columns.

```
df = spark.readStream\
          .format("cloudFiles")\
          .option("cloudFiles.format", "csv")\
          .option("cloudFiles.partitionColumns","year,month,day")\
```

```
            .option("cloudFiles.schemaLocation", "<schema_location>")\
            .option("cloudFiles.useIncrementalListing", "auto")\
            .load("<root_path>")
```

```
df.display()
```

▸ (1) Spark Jobs

▸ display_query_1 (id: 7094ca6d-9676-43cd-b888-d91c8755e45f) *Last updated: 2 hours ago*

Table +

	Name	Number	year	month	day	_rescued_data
1	pr1	1234	2022	january	21	null
2	pr2	647	2022	january	21	null
3	pr9	2734	2023	july	20	null
4	pr10	2744	2023	july	20	null
5	pr11	748	2023	october	12	null
6	pr12	7572	2023	october	12	null

If these columns are not required as part of the schema, “” can be specified to hide these columns.

```
df = spark.readStream\
            .format("cloudFiles")\
            .option("cloudFiles.format", "csv")\
            .option("cloudFiles.partitionColumns","")\
            .option("cloudFiles.schemaLocation", "<schema_location>")\
            .option("cloudFiles.useIncrementalListing", "auto")\
            .load("<root_path>")
```

```
df =spark.readStream\
        .format("cloudFiles")\
        .option("cloudFiles.format", "csv")\
        .option("cloudFiles.partitionColumns","")\
        .option("cloudFiles.schemaLocation", "<schema_location>")\
        .option("cloudFiles.useIncrementalListing", "auto")\
        .load("<location_of_root_folder>")
```

df: pyspark.sql.dataframe.DataFrame = [Name: string, Number: string ... 1 more field]

Command took 0.35 seconds -- by rahul.singha@databricks.com at 10/02/2023, 11:10:29 on test-rahul

Cmd 8

```
df.display()
```

(1) Spark Jobs

display_query_4 (id: 3dedaed1-9cb2-4b25-a20f-d99c546a29e0) *Last updated: 1 minute ago*

Table +

	Name	Number	_rescued_data
1	pr5	3743	null
2	pr6	2542	null
3	pr3	374	null
4	pr4	5763	null
5	pr1	1234	null
6	pr2	647	null

Showing all 12 rows.

5. cloudFiles.allowOverwrites

In Databricks, autoloader by default does not process a file if it is processed once even if the file is modified.

In order to solve this problem, cloudFiles.allowOverwrites option is used to process the files again if the files have been modified.

This option is available in Databricks Runtime 7.6 and above. The default value is false.

This may result in another problem. If files are modified and processed again by the autoloader, duplicate records will be generated.

To tackle this problem, merge insert can be used in each micro batch using Upsert class.

```
sql_query = """
  MERGE INTO table_upsert_update  a
  USING stream_updates b
  ON a.Name=b.Name
  WHEN MATCHED AND a.Number!=b.Number THEN UPDATE SET *
  WHEN NOT MATCHED THEN INSERT *
"""

class Upsert:
    def __init__(self, sql_query, update_temp="stream_updates"):
        self.sql_query = sql_query
        self.update_temp = update_temp

    def upsert_to_delta(self, microBatchDF, batch):
        microBatchDF.createOrReplaceTempView(self.update_temp)
        microBatchDF._jdf.sparkSession().sql(self.sql_query)

#passing the sql_query to Upsert class
streaming_merge = Upsert(sql_query)
```

Since SQL is being used to write to the Delta table, it is required to create the table before using it.

```
%sql
CREATE TABLE IF NOT EXISTS table_upsert_update
```

```
(Name String, Number String)
USING DELTA
```

Now, if the streaming operation is performed with the Autoloader keeping the allowOverwrite as True, records can be obtained from the overwritten file without allowing duplicate records.

```
query = (spark.readStream\
                .format("cloudFiles")\
                .option("cloudFiles.format", "csv")\
                .option("cloudFiles.allowOverwrites",True)\
                .option("cloudFiles.schemaLocation", "<schema_location>")\
                .option("cloudFiles.useIncrementalListing", "auto")\
                .load("<file_location>")\
                .writeStream
                .foreachBatch(streaming_merge.upsert_to_delta)
                .outputMode("update")
                .option("checkpointLocation", "<checkpoint_location>")\
                .start())
```

The records will be stored in table_upsert_update table.

However, Files are processed exactly once unless cloudFiles.allowOverwrites is enabled. If a file is appended to or overwritten, Databricks does not guarantee which version of the file is processed. It is, therefore, recommended to use Auto Loader to ingest only immutable files.

6. ignoreMissingFiles

Autoloader loads files in two steps. In the first step, Autoloader records all the files in the checkpoint directory. This is followed by the actual processing of files from the list of files in the checkpoint directory..

If the file is removed or deleted before the Autoloader processes the file, an error or exception is raised and the streaming stops.

In order to avoid this exception and continue the streaming uninterrupted, the option ignoreMissingFiles is present in Autoloader.

If this option is set to true, the Spark jobs will continue to run when encountering missing files and the contents that have been read will still be returned. Available in Databricks Runtime 11.0 and above.

The default value is False (True for COPY INTO).

```
spark.readStream\
     .format("cloudFiles")\
     .option("cloudFiles.format", "csv")\
     .option("ignoreMissingFiles",True)\
     .option("cloudFiles.schemaLocation", "<schema_location>")\
     .option("cloudFiles.useIncrementalListing", "auto")\
     .load("<files_location>")\
     .writeStream\
     .option("checkpointLocation", "<checkpoint_directory>")\
     .table("table_name>")
```

7. pathGlobFilter

pathGlobFilter is used to only include or exclude files with file names matching the pattern. The syntax follows org.apache.hadoop.fs.GlobFilter. It does not change the behavior of partition discovery. **Note: This is validated only for file listing mode.**

```
df=spark.readStream\
        .format("cloudFiles")\
        .option("cloudFiles.format", "csv")\
        .option("cloudFiles.schemaLocation", "<schema_location>")\
        .option("cloudFiles.useIncrementalListing", "auto")\
        .option("pathGlobfilter","*pattern*")\
        .load("<file_location>")
```

This will process files with the following pattern in the file name.

In pathGlobfilter,

- ? matches only a single character
- * matches zero or more than one character.
- ^ Matches a single character that is not from the character set or range {a}. Note that the ^ character must occur immediately to the right of the opening bracket.

Pattern	Description
pattern	Process files if the given pattern is present in the file name
*.csv	Processes only CSV files
{pattern1,pattern2}	Process files if the given patterns are present in the file name
{p1,p2{p3,p4}}	Process files when {p1, p2p3, p2p4} is present in the file name.

pathGlobfilter can only work with the file names and not with the file path and directories.

pathGlobfilter can also be used in the path e.g., *.load('/source-path/2022/06/13/05*/'),* or as a combination of both.

In order to attain filtering based on a particular path or directories, **_metadata** column can be used to filter post-landing of data in the sink. For example

df.select("","_metadata").select("*","_metadata.file_path").filter("file_path not REGEXP '/folder_name/' ").display()*

This will exclude all files from the given folder_name. However, this is just a simple filter (using SQL REGEXP) applied to the whole data frame and excludes data post-landing in Autoloader.

This can be easily understood with the help of an example.

Let's assume that there are some files in a directory containing sales data for shop1, shop2, and online sales.

```
display(dbutils.fs.ls("<file_location>"))
```

▸ (4) Spark Jobs

Table +

	path		name	size	modificationTime
1	<file_location>	/online_01012023.csv	online_01012023.csv	54	1678992892000
2	<file_location>	/shop1_01012023.csv	shop1_01012023.csv	52	1678992906000
3	<file_location>	/shop1_02012023.csv	shop1_02012023.csv	52	1678992915000
4	<file_location>	/shop2_01012023.csv	shop2_01012023.csv	54	1678992924000
5	<file_location>	/shop2_02012023.csv	shop2_02012023.csv	56	1678992196000

5 rows | 1.04 seconds runtime

Now if it is required to process files of only shop1, the following code can be used.

```
df_shop1=spark.readStream\
        .format("cloudFiles")\
        .option("cloudFiles.format", "csv")\
        .option("cloudFiles.schemaLocation", "<schema_location>")\
        .option("cloudFiles.useIncrementalListing", "auto")\
        .option("pathGlobfilter","*{shop1}*")\
        .load("<file_location>")
```

This will load data only from files that have the pattern “shop1” in the file name which can be verified by displaying the metadata column of the dataframe.

Similarly, if to load data of shop1 and online sales,

“*{shop1,online}*” can be used in pathGlobfilter.

Now, if it is required to load data from all files except certain files, let’s say, sales data from shop2, pathGlobfilter cannot be used. In such cases, SQL

REGEXP can be used.

```
df_notshop2 =spark.readStream\
                  .format("cloudFiles")\
                  .option("cloudFiles.format", "csv")\
                  .option("cloudFiles.schemaLocation", "<schema_location>")\
                  .option("cloudFiles.useIncrementalListing", "auto")\
                  .load("<file_location>")\
                  .select("*","_metadata").select("*","_metadata.file_path")\
                  .filter("file_path not REGEXP 'shop2' ")
```

8. Moving Autoloader Job from one Workspace to another to Another

Autoloader jobs can be easily moved from one workspace to another by just replicating the same code in the target workspace and directing the checkpoint to the same checkpoint used before.

If the checkpoint point location is also changed, the Autoloader will start from the beginning and there may be duplicate records in the sink.

An example will make it more clear.

Autoloader is used to load files from a source with the code as shown below:

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

```
        .option("cloudFiles.schemaLocation", "<schema_location>")\
        .option("cloudFiles.useIncrementalListing", "auto")\
        .load("<location_of_files>")\
        .withColumn("Processing_timestamp", current_timestamp())\
        .writeStream\
        .option("checkpointLocation", "<initial_checkpoint_location>")\
        .trigger(availableNow=True)\
        .table("table_move")
```

In this sample of code, a column “Processing_timestamp” is created which will store the timestamp when the record is processed by the autoloader.

```
%sql
select * from table_move
```

▸ (2) Spark Jobs

▸ _sqldf: pyspark.sql.dataframe.DataFrame = [Name: string, Number: string ... 2 more fields]

Table +

	Name	Number	_rescued_data	Processing_timestamp
1	pr1	1234	null	2023-03-16T21:46:11.043+0000
2	pr2	364	null	2023-03-16T21:46:11.043+0000
3	pr3	731	null	2023-03-16T21:46:11.043+0000

3 rows | 0.38 seconds runtime

New files are then loaded in the source location.

Now, the autoloader code is moved to another workspace with a new checkpoint location and the autoloader is fired. It is observed that duplicate records are generated in the sink, table_move. The Processing_timestamp column signifies that the old files are processed again along with the new files.

```
%sql
select * from table_move
```

▸ (3) Spark Jobs

▸ _sqldf: pyspark.sql.dataframe.DataFrame = [Name: string, Number: string ... 2 more fields]

Table +

	Name	Number	_rescued_data	Processing_timestamp
1	pr2	364	null	2023-03-16T21:46:11.043+0000
2	pr3	731	null	2023-03-16T21:46:11.043+0000
3	pr1	1234	null	2023-03-16T21:46:11.043+0000
4	pr4	873	null	2023-03-16T21:53:03.743+0000
5	pr1	1234	null	2023-03-16T21:53:03.743+0000
6	pr3	731	null	2023-03-16T21:53:03.743+0000
7	pr2	364	null	2023-03-16T21:53:03.743+0000

Duplicate Records

7 rows | 0.52 seconds runtime

If it is required to resume the Autoloader from where it stopped last to avoid duplicate records in the sink, two approaches can be followed:

1. the autoloader should point to the old checkpoint location
2. the files of the old checkpoint location should be copied to the new checkpoint location.

However, after changing the checkpoint location, the autoloader code can be modified to perform only stateless transformations (like filter). If stateful transformations (like groupBy) are added to the code, an exception will be thrown.

```
spark.readStream\
    .format("cloudFiles")\
    .option("cloudFiles.format", "csv")\
    .option("cloudFiles.schemaLocation", "<schema_location>")\
    .option("cloudFiles.useIncrementalListing", "auto")\
    .load("<file_location>")\
    .withColumn("curr_timestamp", current_timestamp())\
    .groupBy("Name").agg(sum("Number").alias("Sum"))\
    .writeStream\
    .option("checkpointLocation", "<new_checkpointlocation>")\
    .trigger(availableNow=True)\
    .table("table_move")
```

Python

AnalysisException: Append output mode not supported when there are streaming aggregations on streaming DataFrames/DataSets without watermark;
Aggregate [Name#13119], [Name#13119, sum(cast(Number#13120 as double)) AS Sum#13136]
+- Project [Name#13119, Number#13120, _rescued_data#13121, current_timestamp() AS curr_timestamp#13125]
   +- StreamingRelation DataSource(org.apache.spark.sql.SparkSession@3042d2fc,cloudFiles,List(),None,List(),None,Map(cloudFiles.format -> csv, cloudFiles.schemaLocation -> abfss://move2@rahultestcodec.dfs.core.windows.net/autoloaderData/schema, cloudFiles.useIncrementalListing -> auto, path -> abfss://move@rahultestcodec.dfs.core.windows.net/data),None), cloudFiles, [Name#13119, Number#13120, _rescued_data#13121]

Command took 1.78 seconds -- by rahul.singha@databricks.com at 17/03/2023, 03:43:28 on test-rahul

Autoloader | Spark Streaming | Databricks | Data Engineering | Data Engineering 101

Written by Rahul Singha

24 followers · 4 following

Follow

Responses (1)

Write a response

What are your thoughts?

Brett M

May 6, 2023

Rahul,

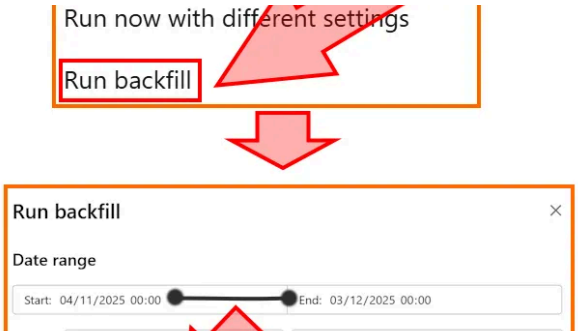
This Databricks article was excellent about Autoloader with great examples. I hope you write more Databricks articles!

Thanks,

Brett

5 Reply

Recommended from Medium

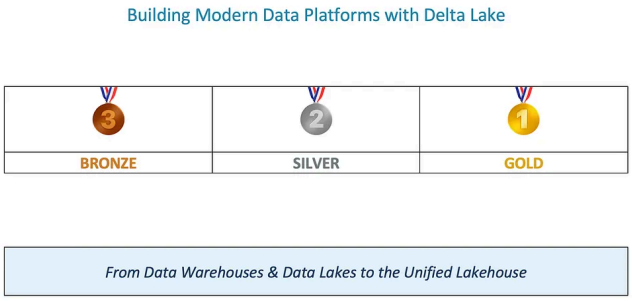


Hubert Dudek

Databricks Lakeflow Jobs Workflow Backfill

In modern data pipelines, incremental processing is the standard approach for...

6d ago 9



In EndToEndData by Prem Vishnoi(cloudvala)

The Complete Guide To Lakehouse Architecture

1.2 Data Lakes: Scale Without Trust

Dec 30, 2025 75

Pallavi Sinha

Powering SQL with AI Functions in Databricks

Namaste Databricks

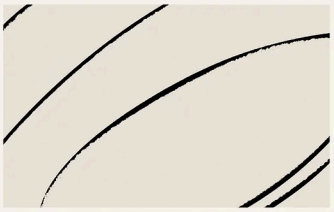
Why Databricks Lakehouse?

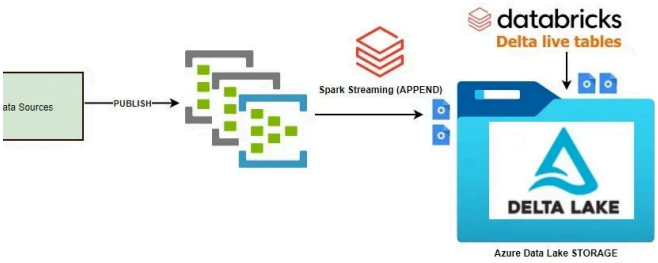
Storage is the foundation of all data systems, providing the means to store, organise and...

Data engineers and analysts spend much of their time in SQL—cleaning data,...

Jul 29, 2025 24 1

Oct 11, 2025





Umesh Pawar

Delta Lake Change Data Feed (CDF) Reduces ETL Costs and...

GitHub repository: umeshpawar2188/Delta-change-data-feed

Jul 25, 2025

Tharaniesh

Building a Real Time Lakehouse Data Platform (LDP) with Delta Li...

From Raw Streams to Business Insights: My Hands On Journey with Medallion...

Nov 18, 2025 1

See more recommendations