Data Engineering

Join discussions on data engineering best practices, architectures, and optimization strategies within the Databricks Community. Exchange insights and solutions with fellow data engineers.

Databricks Community › **Data Engineering** › Re: Auto Loader for copying files on s3 Options

Auto Loader for copying files on s3

✓ Go to solution

daan_dw

New Contributor III

05-08-2025 08:20 AM

Hey community,

I have a folder on s3 with around 5 million small files. On a daily basis new files are added. I would like to simply copy those new files to another folder on s3. My approach is to use an Auto Loader of which I attached the code below. The code works but is too slow. Is there any way to speed up this process? Or is there another approach without Auto Loader that is faster?

Thanks a lot!

```
from pyspark.sql.functions import col, lit
import datetime

source_path = "s3://"
destination_path = "s3://"
checkpoint_path = "/tmp/autoloader_checkpoints/test_job"
schema_path = "/tmp/autoloader_schemas/test_job"


stream_df = (spark.readStream
    .format("cloudFiles")
    .option("cloudFiles.format", "binaryFile")
    .option("cloudFiles.schemaLocation", schema_path)
    .option("cloudFiles.includeExistingFiles", "false")
    .option("recursiveFileLookup", "true")
    .option("pathGlobFilter", "*")
    .option("cloudFiles.useNotifications", "true")
    .option("cloudFiles.region", "eu-central-1")
    .load(source_path))

# Process each file while preserving directory structure
def copy_files(batch_df, batch_id):
    for row in batch_df.collect():
        try:
            src_path = row['path']
            relative_path = src_path.replace(source_path, "")
            dest_path = destination_path + relative_path

            parent_dir = "/".join(dest_path.split("/")[:-1])
            dbutils.fs.mkdirs(parent_dir)
            dbutils.fs.cp(src_path, dest_path)
        except Exception as e:
            print(f"Failed to copy {src_path}: {str(e)}")

(stream_df.writeStream
    .foreachBatch(copy_files)
    .option("checkpointLocation", checkpoint_path)
    .trigger(availableNow=True)
    .start()
    .awaitTermination())
```

0 Kudos

Reply

All forum topics ‹ Previous Topic Next Topic ›

1 ACCEPTED SOLUTION

✓

daan_dw

New Contributor III

05-12-2025 01:29 AM

☑ Hey LRALVA

The first time running your code I got the error: *PicklingError: Could not serialize object: Exception: You cannot use dbutils within a spark job You cannot use dbutils within a spark job or otherwise pickle it.*

So I changed the copy_single_file function to the one below and now it works exactly as expected. Thanks for putting me on the right track!

```
from pyspark.sql.functions import col, lit, input_file_name, regexp_replace, regexp_extract, concat
from pyspark.sql.types import BooleanType, StringType
from pyspark import SparkFiles
import os
import time
import boto3

source_path = ""
destination_path = ""
checkpoint_path = ""
schema_path = ""


spark.conf.set("spark.sql.files.maxPartitionBytes", "128m") # Smaller partitions for more parallelism
spark.conf.set("spark.sql.adaptive.enabled", "true") # Enable adaptive query execution
spark.conf.set("spark.default.parallelism", 100) # Adjust based on your cluster size
spark.conf.set("spark.sql.shuffle.partitions", 100) # Adjust based on your cluster size
spark.conf.set("spark.databricks.io.cache.enabled", "true") # Enable IO cache if on Databricks


def copy_single_file(src_path, dest_path):
    try:
        s3 = boto3.client('s3')

        # Strip 's3://' and split into bucket and key
        def split_s3_path(s3_path):
            s3_path = s3_path.replace("s3://", "")
            bucket = s3_path.split("/")[0]
            key = "/".join(s3_path.split("/")[1:])
            return bucket, key

        source_bucket, source_key = split_s3_path(src_path)
        dest_bucket, dest_key = split_s3_path(dest_path)

        s3.copy_object(
            CopySource={'Bucket': source_bucket, 'Key': source_key},
            Bucket=dest_bucket,
            Key=dest_key
        )

        return "success"
    except Exception as e:
        return f"error: {str(e)}"

copy_file_udf = udf(copy_single_file, StringType())

file_stream = (spark.readStream
.format("cloudFiles")
.option("cloudFiles.format", "binaryFile") # Read files in binary format
.option("cloudFiles.schemaLocation", schema_path)
.option("cloudFiles.includeExistingFiles", "true")
.option("recursiveFileLookup", "true") # Search all subdirectories
.option("pathGlobFilter", "*") # Process all file types
.option("cloudFiles.useNotifications", "true") # Use S3 notifications if available
.option("cloudFiles.fetchParallelism", 64) # Increase parallelism for listing files
.option("cloudFiles.maxFilesPerTrigger", 10000) # Process more files per batch
.option("cloudFiles.region", "eu-central-1")
.load(source_path)
.select("path", "length", "modificationTime")) # Only select needed columns to reduce memory

def process_batch(batch_df, batch_id):
    start_time = time.time()
    if batch_df.count() == 0:
        print(f"Batch {batch_id}: No files to process")
        return
    timestamp = time.strftime("%Y-%m-%d %H:%M:%S")
    processed_df = (batch_df
    # Create destination path by replacing source path with destination path
    .withColumn("relative_path",
    regexp_replace("path", source_path, ""))
    .withColumn("destination_path",
    concat(lit(destination_path), col("relative_path")))
    # Apply the copy operation to each file in parallel
    .withColumn("copy_result",
    copy_file_udf(col("path"), col("destination_path")))
    )
    file_count = batch_df.count()
    optimal_partitions = min(max(file_count // 1000, 8), 128) # Between 8-128 pa
```

```
rtitions
    result_df = processed_df.repartition(optimal_partitions).cache()
    success_count = result_df.filter(col("copy_result").startswith("success")).c
ount()
    error_count = result_df.filter(col("copy_result").startswith("error")).count
()
    if error_count > 0:
        errors_df = result_df.filter(col("copy_result").startswith("error"))
        print(f"Batch {batch_id}: Found {error_count} errors. Sample errors:")
        errors_df.select("path", "copy_result").show(10, truncate=False)
    duration = time.time() - start_time
    files_per_second = file_count / duration if duration > 0 else 0

    # Log summary
    print(f"""
    Batch {batch_id} completed at {timestamp}:
    - Files processed: {file_count}
    - Success: {success_count}
    - Errors: {error_count}
    - Duration: {duration:.2f} seconds
    - Performance: {files_per_second:.2f} files/second
    """)
    result_df.unpersist()

(file_stream.writeStream
.foreachBatch(process_batch)
.option("checkpointLocation", checkpoint_path)
.trigger(availableNow=True) # Process available files and terminate
.start()
.awaitTermination())
```

View solution in original post

0 Kudos

Reply

3 REPLIES

lingareddy_Alva

Honored Contributor III

05-08-2025 01:47 PM - edited 05-08-2025 01:49 PM

@daan_dw

You're facing a classic small files problem in S3, which is challenging to solve efficiently. Your current Auto Loader approach has performance limitations when processing millions of small files individually. Let me suggest several optimizations and alternative approaches to speed up this process.
Key Performance Issues

Your copy_files function processes files one by one with dbutils.fs.cp, creating a lot of overhead
Using batch_df.collect() brings all file paths to the driver, creating memory pressure
Individual S3 operations have high latency, especially when done sequentially.

Try below code:

```
from pyspark.sql.functions import col, lit, input_file_name, regexp_replace, regexp_extract
from pyspark.sql.types import BooleanType, StringType
from pyspark import SparkFiles
import os
import time

# S3 paths configuration
source_path = "s3://source-bucket/source-folder/"
destination_path = "s3://destination-bucket/destination-folder/"
checkpoint_path = "/tmp/autoloader_checkpoints/file_copy_job"
schema_path = "/tmp/autoloader_schemas/file_copy_job"
```

```
# Performance tuning configuration
spark.conf.set("spark.sql.files.maxPartitionBytes", "128m") # Smaller partitions for more parallelism
spark.conf.set("spark.sql.adaptive.enabled", "true") # Enable adaptive query execution
spark.conf.set("spark.default.parallelism", 100) # Adjust based on your cluster size
spark.conf.set("spark.sql.shuffle.partitions", 100) # Adjust based on your cluster size
spark.conf.set("spark.databricks.io.cache.enabled", "true") # Enable IO cache if on Databricks

# Define a UDF for copying files that preserves structure and handles errors
def copy_single_file(src_path, dest_path😞
try:
# Ensure parent directory exists
parent_dir = "/".join(dest_path.split("/")[:-1]) + "/"
dbutils.fs.mkdirs(parent_dir)

# Copy the file
dbutils.fs.cp(src_path, dest_path)
return "success"
except Exception as e:
return f"error: {str(e)}"

# Register the UDF
copy_file_udf = udf(copy_single_file, StringType())

# Set up the Auto Loader stream
file_stream = (spark.readStream
.format("cloudFiles")
.option("cloudFiles.format", "binaryFile") # Read files in binary format
.option("cloudFiles.schemaLocation", schema_path)
.option("cloudFiles.includeExistingFiles", "false")
.option("recursiveFileLookup", "true") # Search all subdirectories
.option("pathGlobFilter", "*") # Process all file types
.option("cloudFiles.useNotifications", "true") # Use S3 notifications if available
```

```
.option("cloudFiles.fetchParallelism", 64) # Increase parallelism for listing files
.option("cloudFiles.maxFilesPerTrigger", 10000) # Process more files per batch
.option("cloudFiles.region", "eu-central-1")
.load(source_path)
.select("path", "length", "modificationTime")) # Only select needed columns to reduce
memory

# Process each batch efficiently
def process_batch(batch_df, batch_id😞
start_time = time.time()

# Skip empty batches
if batch_df.count() == 0:
print(f"Batch {batch_id}: No files to process")
return

# Get timestamp for logging
timestamp = time.strftime("%Y-%m-%d %H:%M:%S")

# Calculate the destination path for each file by preserving directory structure
processed_df = (batch_df
# Create destination path by replacing source path with destination path
.withColumn("relative_path",
regexp_replace("path", source_path, ""))
.withColumn("destination_path",
concat(lit(destination_path), col("relative_path")))
# Apply the copy operation to each file in parallel
.withColumn("copy_result",
copy_file_udf(col("path"), col("destination_path")))
)

# Repartition for better parallelism based on file size distribution
# More partitions = more parallel operations
file_count = batch_df.count()
optimal_partitions = min(max(file_count // 1000, 8), 128) # Between 8-128 partitions
```

```
# Force execution and collect metrics
result_df = processed_df.repartition(optimal_partitions).cache()

# Trigger execution and collect stats
success_count = result_df.filter(col("copy_result").startswith("success")).count()
error_count = result_df.filter(col("copy_result").startswith("error")).count()

# Log errors for investigation
if error_count > 0:
errors_df = result_df.filter(col("copy_result").startswith("error"))
print(f"Batch {batch_id}: Found {error_count} errors. Sample errors:")
errors_df.select("path", "copy_result").show(10, truncate=False)

# Optionally write errors to a log location
errors_df.write.mode("append").parquet(f"{destination_path}/_error_logs/{batch_id}")

# Calculate performance metrics
duration = time.time() - start_time
files_per_second = file_count / duration if duration > 0 else 0

# Log summary
print(f"""
Batch {batch_id} completed at {timestamp}:
- Files processed: {file_count}
- Success: {success_count}
- Errors: {error_count}
- Duration: {duration:.2f} seconds
- Performance: {files_per_second:.2f} files/second
""")

# Unpersist to free memory
result_df.unpersist()

# Execute the streaming job
```

```
(file_stream.writeStream
.foreachBatch(process_batch)
.option("checkpointLocation", checkpoint_path)
.trigger(availableNow=True) # Process available files and terminate
.start()
.awaitTermination())
```

LR

👍 0 Kudos

Reply

✓

daan_dw

New Contributor III

05-12-2025 01:29 AM

☑ Hey LRALVA

The first time running your code I got the error: *PicklingError: Could not serialize object: Exception: You cannot use dbutils within a spark job You cannot use dbutils within a spark job or otherwise pickle it.*

So I changed the copy_single_file function to the one below and now it works exactly as expected. Thanks for putting me on the right track!

```
from pyspark.sql.functions import col, lit, input_file_name, regexp_replace, regexp_extract, concat
from pyspark.sql.types import BooleanType, StringType
from pyspark import SparkFiles
import os
import time
import boto3

source_path = ""
destination_path = ""
checkpoint_path = ""
schema_path = ""


spark.conf.set("spark.sql.files.maxPartitionBytes", "128m") # Smaller partitions for more parallelism
spark.conf.set("spark.sql.adaptive.enabled", "true") # Enable adaptive query execution
spark.conf.set("spark.default.parallelism", 100) # Adjust based on your cluster size
spark.conf.set("spark.sql.shuffle.partitions", 100) # Adjust based on your cluster size
spark.conf.set("spark.databricks.io.cache.enabled", "true") # Enable IO cache if on Databricks


def copy_single_file(src_path, dest_path):
    try:
        s3 = boto3.client('s3')

        # Strip 's3://' and split into bucket and key
        def split_s3_path(s3_path):
            s3_path = s3_path.replace("s3://", "")
            bucket = s3_path.split("/")[0]
            key = "/".join(s3_path.split("/")[1:])
            return bucket, key

        source_bucket, source_key = split_s3_path(src_path)
        dest_bucket, dest_key = split_s3_path(dest_path)

        s3.copy_object(
            CopySource={'Bucket': source_bucket, 'Key': source_key},
            Bucket=dest_bucket,
            Key=dest_key
        )

        return "success"
    except Exception as e:
        return f"error: {str(e)}"

copy_file_udf = udf(copy_single_file, StringType())

file_stream = (spark.readStream
.format("cloudFiles")
.option("cloudFiles.format", "binaryFile") # Read files in binary format
.option("cloudFiles.schemaLocation", schema_path)
.option("cloudFiles.includeExistingFiles", "true")
.option("recursiveFileLookup", "true") # Search all subdirectories
.option("pathGlobFilter", "*") # Process all file types
.option("cloudFiles.useNotifications", "true") # Use S3 notifications if available
.option("cloudFiles.fetchParallelism", 64) # Increase parallelism for listing files
.option("cloudFiles.maxFilesPerTrigger", 10000) # Process more files per batch
.option("cloudFiles.region", "eu-central-1")
.load(source_path)
.select("path", "length", "modificationTime")) # Only select needed columns to reduce memory

def process_batch(batch_df, batch_id):
    start_time = time.time()
    if batch_df.count() == 0:
        print(f"Batch {batch_id}: No files to process")
        return
    timestamp = time.strftime("%Y-%m-%d %H:%M:%S")
    processed_df = (batch_df
    # Create destination path by replacing source path with destination path
    .withColumn("relative_path",
    regexp_replace("path", source_path, ""))
    .withColumn("destination_path",
    concat(lit(destination_path), col("relative_path")))
    # Apply the copy operation to each file in parallel
    .withColumn("copy_result",
    copy_file_udf(col("path"), col("destination_path")))
    )
    file_count = batch_df.count()
    optimal_partitions = min(max(file_count // 1000, 8), 128) # Between 8-128 pa
```

```
rtitions
    result_df = processed_df.repartition(optimal_partitions).cache()
    success_count = result_df.filter(col("copy_result").startswith("success")).c
ount()
    error_count = result_df.filter(col("copy_result").startswith("error")).count
()
    if error_count > 0:
        errors_df = result_df.filter(col("copy_result").startswith("error"))
        print(f"Batch {batch_id}: Found {error_count} errors. Sample errors:")
        errors_df.select("path", "copy_result").show(10, truncate=False)
    duration = time.time() - start_time
    files_per_second = file_count / duration if duration > 0 else 0

    # Log summary
    print(f"""
    Batch {batch_id} completed at {timestamp}:
    - Files processed: {file_count}
    - Success: {success_count}
    - Errors: {error_count}
    - Duration: {duration:.2f} seconds
    - Performance: {files_per_second:.2f} files/second
    """)
    result_df.unpersist()

(file_stream.writeStream
.foreachBatch(process_batch)
.option("checkpointLocation", checkpoint_path)
.trigger(availableNow=True) # Process available files and terminate
.start()
.awaitTermination())
```

0 Kudos

Reply

lingareddy_Alva

Honored Contributor III

In response to **daan_dw**

05-12-2025 07:33 AM

Hey @daan_dw Thanks for the update.

LR

👍 0 Kudos

Reply

Reply to the topic...

Post Reply

Announcements

🎬 Databricks Community 2025 Highlights | A Year, Built Together

👁 541 💬 12 👍 17

on Wednesday

🌞 Community Pulse: Your Weekly Roundup! December 22, 2025 – January 04, 2026

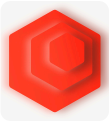
👁 154 💬 2 👍 4

on Tuesday

Solution Accelerator Series | Scale cybersecurity analytics with Splunk and Databricks

👁 374 💬 1 👍 2

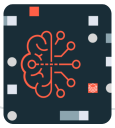
on Tuesday



🎤 Call for Presentations: Data + AI Summit 2026 is Open!

👁 4674 💬 4 👍 6

on 4 weeks ago



Self-Paced Learning Festival: 09 January - 30 January 2026

👁 51227 💬 139 👍 81

on 12-09-2025

Related Content

Is anyone getting up and working ? Federating Snowflake-managed Iceberg tables into Azure Databricks

in **Data Engineering** 2 weeks ago

DLT Autoloader schemaHints from JSON file instead of inline list?

in **Data Engineering** 12-03-2025

Serving model issue in databricks

in **Generative AI** 11-26-2025

Autoloader Managed File events

in **Data Engineering** 11-18-2025

Migrating from directory-listing to Autoloader Managed File events

in **Data Engineering** 11-17-2025

Databricks Inc.
160 Spear Street, 13th
Floor
San Francisco, CA 94105
1-866-330-0121