## Databricks Best Practise & Learning’s

**# Databricks Streaming Pipeline (Step-by-Step)**

# Sample file: people.csv (fields: id, name, department, event\_time)

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

from pyspark.sql.types import StructType, StructField, IntegerType, StringType, TimestampType

# 1. Define Input & Output Paths

source\_path = "/mnt/data/autoloader/incoming/"

schema\_path = "/mnt/data/autoloader/schema/people/"

checkpoint\_path = "/mnt/data/autoloader/checkpoints/people/"

target\_path = "/mnt/data/bronze/people/"

# 2. Define Schema Explicitly (for clarity)

schema = StructType([

StructField("id", IntegerType(), True),

StructField("name", StringType(), True),

StructField("department", StringType(), True),

StructField("event\_time", TimestampType(), True)

])

# 3. Read Files Using Auto Loader (cloudFiles)

1) CSV

df = (

spark.readStream

.format("cloudFiles")

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

.option("cloudFiles.inferColumnTypes", "true")

.option("cloudFiles.schemaLocation", schema\_path)

.option("header", "true")

.load(source\_path)

)

# Handle other formats by checking file extension dynamically

from pyspark.sql.functions import input\_file\_name, expr

df = df.withColumn("format", expr("CASE

WHEN input\_file\_name LIKE '%.json' THEN 'json'

WHEN input\_file\_name LIKE '%.parquet' THEN 'parquet'

ELSE 'csv' END"))

# 4. Apply Watermark (late data tolerance)

df\_watermarked = df.withWatermark("event\_time", "10 minutes")

# 5. (Optional) Time-Based Windowing (for aggregations)

df\_windowed = df\_watermarked.groupBy(

window(col("event\_time"), "30 minutes"),

col("department")

).count()

# 6. Deduplicate Based on id + source file path

df\_deduped = df.dropDuplicates(["id", "name", "event\_time"])

# 7. Transformations (enrichment, metadata stamping)

df\_transformed = df\_deduped \

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

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

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

# 8. Write to Delta Table with Checkpointing

(

df\_transformed.writeStream

.format("delta")

.outputMode("append")

.option("checkpointLocation", checkpoint\_path)

.start(target\_path)

)

# Notes:

# - This pipeline processes incoming CSV files with schema inference

# - Watermark ensures late data (up to 10 minutes) is processed

# - Checkpoints track state for recovery

# - Deduplication avoids duplicates in the Bronze table

The complete step-by-step code has been created as a document titled **"Autoloader Streaming Pipeline"**. It covers:

* Reading files with cloudFiles
* Applying watermark
* Optional windowing
* Deduplication
* Transformations
* Writing to Delta with checkpoint

Notes:

# - This pipeline processes CSV, JSON, and Parquet files

# - File format is inferred from extension and marked in a column

# - Schema inference is enabled for flexibility

# - Watermark handles late data up to 10 minutes

# - Deduplication and enrichment ensure clean, auditable data

Appendix:

## Format-Specific Options

Some options also depend on the file type you're ingesting:

### For CSV:

* header → "true" if the file has headers
* delimiter → "," or "|", etc.
* quote, escape, nullValue, etc.

### For JSON:

* multiline → "true" for JSON lines with nested objects
* allowBackslashEscapingAnyCharacter

### For Parquet/Avro/ORC:

* Typically schema is embedded; fewer format-specific options needed

## Recommended Processing Sequence in Databricks Streaming (Bronze Layer)

|  |  |  |
| --- | --- | --- |
| **Order** | **Component / Feature** | **Purpose** |
| 1 | **cloudFiles** | File discovery engine – detects new files from cloud storage |
| 2 | **Auto Loader** | Wraps cloudFiles and handles schema inference, checkpointing, ingestion |
| 3 | **Schema Inference** | Defines and persists the schema to avoid redundant inferring |
| 4 | **Watermarking** | Manages late-arriving data using event-time thresholds |
| 5 | **Windowing** (if used) | Optional: applies event-time window aggregation |
| 6 | **Deduplication** | Removes duplicate records before writing to Delta |
| 7 | **Transformation** | Column cleanup, enrichment, lineage stamping (e.g., input\_file\_name()) |
| 8 | **Checkpointing** | Tracks progress of ingestion and stream reliability |
| 9 | **Write to Delta** | Append data to Bronze or Silver Delta table |

## Optional or Layer-Specific Steps

|  |  |  |  |
| --- | --- | --- | --- |
| **Step** | **Component** | **Used In** | **Purpose** |
| – | Validation Rules | Silver layer | Clean data using business/data rules |
| – | merge or upsert | Silver/Gold layer | Handle Slowly Changing Dimensions (SCD) |
| – | OPTIMIZE / ZORDER | After write | Improve read performance |
| – | Unity Catalog / ACLs | Governance layer | Apply security, auditing, lineage |

## Final Order (Practical Names)

CopyEdit

CloudFiles → Auto Loader → Schema Inference → Watermark → Windowing (optional) → Deduplication → Transformations → Checkpointing → Write to Delta

Conslidated Code in One Go:

# Databricks Streaming Pipeline (Step-by-Step)

# Multi-format ingestion: CSV, JSON, and Parquet (fields: id, name, department, event\_time)

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

from pyspark.sql.types import StructType, StructField, IntegerType, StringType, TimestampType

# 1. Define Input & Output Paths

source\_path = "/mnt/data/autoloader/incoming/"

schema\_path = "/mnt/data/autoloader/schema/people/"

checkpoint\_path = "/mnt/data/autoloader/checkpoints/people/"

target\_path = "/mnt/data/bronze/people/"

# 2. Define Schema Explicitly (for clarity)

schema = StructType([

StructField("id", IntegerType(), True),

StructField("name", StringType(), True),

StructField("department", StringType(), True),

StructField("event\_time", TimestampType(), True)

])

# 3. Read Files Using Auto Loader (cloudFiles)

df = (

spark.readStream

.format("cloudFiles")

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

.option("cloudFiles.schemaLocation", schema\_path)

.option("cloudFiles.inferColumnTypes", "true")

.option("header", "true")

.load(source\_path)

)

**# Handle file type format tagging**

df = df.withColumn("format", expr("CASE \

WHEN input\_file\_name() LIKE '%.json' THEN 'json' \

WHEN input\_file\_name() LIKE '%.parquet' THEN 'parquet' \

ELSE 'csv' END"))

# 4. Apply Watermark

df\_watermarked = df.withWatermark("event\_time", "10 minutes")

# 5. Optional: Windowed Aggregation (not written to target here)

df\_windowed = df\_watermarked.groupBy(

window(col("event\_time"), "30 minutes"),

col("department")

).count()

# 6. Deduplication

df\_deduped = df.dropDuplicates(["id", "name", "event\_time"])

# 7. Transformations

df\_transformed = df\_deduped \

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

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

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

# 8. Write to Delta Bronze Table

(

df\_transformed.writeStream

.format("delta")

.outputMode("append")

.option("checkpointLocation", checkpoint\_path)

.start(target\_path)

)

# 9. Register Bronze Table

spark.sql("""

CREATE TABLE IF NOT EXISTS bronze\_people

USING DELTA

LOCATION '/mnt/data/bronze/people/'

""")

# 10. Create Silver Table (refined & cleaned)

spark.sql("""

CREATE OR REPLACE TABLE silver\_people AS

SELECT id, name, department, event\_time, ingestion\_timestamp

FROM bronze\_people

WHERE department IS NOT NULL AND name IS NOT NULL

""")

# 11 . Create Gold (curated) Table for ML or Reporting

spark.sql("""

CREATE OR REPLACE TABLE gold\_people\_summary AS

SELECT department, COUNT(\*) AS employee\_count, MAX(event\_time) AS last\_event

FROM silver\_people

GROUP BY department

""")

# 12 . Train ML Model Using Curated Table

from databricks.feature\_store import FeatureLookup, FeatureStoreClient

from pyspark.ml.classification import LogisticRegression

from pyspark.ml.feature import VectorAssembler

# Load curated data

gold\_df = spark.table("gold\_people\_summary")

# Create features

assembler = VectorAssembler(inputCols=["employee\_count"], outputCol="features")

training\_df = assembler.transform(gold\_df).withColumn("label", expr("CASE WHEN department = 'HR' THEN 1 ELSE 0 END"))

# Train simple model

lr = LogisticRegression(maxIter=10)

model = lr.fit(training\_df)

# Save model for dashboard scoring

model.write().overwrite().save("/mnt/models/hr\_classifier")

# Model is now ready for batch or real-time scoring and reporting use

## File Handling Functions in PySpark

### Reading Files

* spark.read.format("csv"), "json", "parquet", etc.
* spark.readStream.format("cloudFiles") (Auto Loader)
* .option("header", "true"), .option("inferSchema", "true")
* .load(path)

### Writing Files

* .write.format("delta"/"parquet"/"csv")
* .writeStream.format("delta")
* .save(path)
* .saveAsTable(tableName)
* .option("checkpointLocation", path)

### File Metadata Functions

* input\_file\_name()
* current\_input\_file\_name()
* spark.\_jsparkSession.catalog.listFiles() (via JVM)

## Delta Table Handling

### Table Operations

* spark.sql("CREATE TABLE ... USING DELTA LOCATION ...")
* spark.read.table("table\_name")
* spark.table("table\_name")
* spark.catalog.listTables()

### Delta Table Management (Python API)

* DeltaTable.forPath(spark, path)
* .update(), .delete(), .merge()
* .history(), .vacuum(), .optimize(), .restoreToVersion()

## Metadata and Catalog

### Table Metadata

* spark.catalog.listTables()
* spark.catalog.listDatabases()
* spark.catalog.listColumns("table\_name")
* spark.catalog.isCached("table\_name")

### Table Registration

* CREATE TABLE USING DELTA LOCATION ...
* CREATE OR REPLACE TABLE AS SELECT ...
* DROP TABLE IF EXISTS ...

## Partitioning and Bucketing

* .partitionBy("column")
* .bucketBy(numBuckets, "column") (limited streaming support)

## Most Common .option() Settings by Context

General for spark.read / spark.readStream

|  |  |  |
| --- | --- | --- |
| **Option** | **Used With** | **Purpose** |
| "header" | CSV | Whether the first row is a header |
| "inferSchema" | CSV, JSON | Infers column types automatically |
| "delimiter" | CSV | Field delimiter (e.g., ",", `" |
| "multiline" | JSON | Handles multi-line JSON |
| "quote" / "escape" | CSV | For quoted fields and escapes |
| "mode" | All formats | e.g., PERMISSIVE, FAILFAST, DROPMALFORMED |
| "path" | All | Explicitly set the file path to read |

## Format-Specific Options

Some options also depend on the file type you're ingesting:

### For CSV:

* header → "true" if the file has headers
* delimiter → "," or "|", etc.
* quote, escape, nullValue, etc.

### For JSON:

* multiline → "true" for JSON lines with nested objects
* allowBackslashEscapingAnyCharacter

### For Parquet/Avro/ORC:

* Typically schema is embedded; fewer format-specific options needed

### For cloudFiles / Auto Loader

|  |  |
| --- | --- |
| **Option** | **Purpose** |
| "cloudFiles.format" | File format (e.g., "csv", "json", etc.) |
| "cloudFiles.schemaLocation" | Path to persist inferred schema |
| "cloudFiles.inferColumnTypes" | Automatically detect types (CSV/JSON) |
| "cloudFiles.includeExistingFiles" | Include already-present files at startup |
| "cloudFiles.useNotifications" | Use cloud notifications (instead of listing) |
| "cloudFiles.maxFilesPerTrigger" | Control batch size by file count |
| "cloudFiles.maxBytesPerTrigger" | Control batch size by total size |

## Most Common .option("cloudFiles.\*") Settings

|  |  |
| --- | --- |
| **Option** | **Description** |
| **cloudFiles.format** | Required. Specifies the file format: csv, json, parquet, avro, orc, text, binaryFile |
| **cloudFiles.schemaLocation** | Required. Stores inferred or user-defined schema persistently for efficient re-use |
| **cloudFiles.inferColumnTypes** | Auto-detects data types from files (for CSV and JSON). Defaults to false |
| **cloudFiles.includeExistingFiles** | Ingests already-existing files when the stream starts. Set to true to include historical files |
| **cloudFiles.useNotifications** | Enables event-based discovery (via S3, ADLS Gen2, or GCS) instead of directory listing |
| **cloudFiles.maxBytesPerTrigger** | Limits total size (in bytes) processed per micro-batch |
| **cloudFiles.maxFilesPerTrigger** | Limits number of new files per micro-batch (default: unlimited) |
| **cloudFiles.backfillInterval** | Controls rate of discovery for old files when includeExistingFiles=true |
| **cloudFiles.partitionColumns** | Used to specify partition columns (especially for Hive-style partitions) |
| **cloudFiles.resourceGroup** | Azure-specific: used for Event Grid setup |
| **cloudFiles.subscriptionId** | Azure-specific: used for Event Grid setup |
| **cloudFiles.connectionString** | Azure-specific: connection to Event Grid topic for file notifications |
| **cloudFiles.validateOptions** | Validates all options at runtime. Default is true |

### For df.write or df.writeStream

|  |  |
| --- | --- |
| **Option** | **Purpose** |
| "checkpointLocation" | Path to track streaming progress |
| "path" | Output path when not using .save(path) directly |
| "compression" | Set output compression (e.g., snappy, gzip) |
| "mergeSchema" | Enable schema evolution in Delta writes |
| "overwriteSchema" | Replace existing table schema (if compatible) |
| "partitionBy" | Partition output files or tables |
| "outputMode" | For streaming: append, update, complete |

### Delta-specific Options

|  |  |
| --- | --- |
| **Option** | **Use Case** |
| "mergeSchema" | Append with evolving schema |
| "overwriteSchema" | Overwrite table schema on write |
| "delta.columnMapping.mode" | Enable column rename/reorder (name, id, none) |